CONV2D ACCELERATION USING CFU-PLAYGROUND

Sai Gautham Ravipati,
Department of Electrical Engineering,
Indian Institute of Technology, Madras.
sai.gautham@smail.iitm.ac.in

1 Introduction

Neural Networks form an integral part of several ML applications, and the convolution operation plays a crucial role during the inference of Neural Networks in recognition of several important features. In the case of 2D convolution, there exists several aspects for parallelism for computation, like independent channels, successive strides etc., which are to be exploited. We also need better data-flow to gain from the parallelism, in the below case of Fig-1, six elements would be common between successive strides, hence, better data-flow shall provide memory re-use. On these grounds, several frameworks have been developed for the hardware acceleration of Deep Neural Networks some of which require the user to build the accelerator, while the rest generate the hardware using templates based on constraints like degree of parallelism. A brief review of existing frameworks has been summarised later. One such framework, CFU Playground[1], has been used in this work to accelerate 3x3 convolutions of an int8 quantised MNIST Neural Network. The framework allows the development of Custom Function Units, CFUs, similar to the case of ALUs to perform specialised operations like Multiply and Accumulate, and these CFUs can be invoked from the TFLite kernels using custom instructions in software which are translated to assembly using compiler directives. Since, a kernel could be used in any Neural Network, the framework allows the usage of same hardware for multiple Neural Networks unlike the case of having the whole network on hardware, which has its limitations in terms of re-use of hardware. In the following sections, a brief review about the framework is presented, post which, a CFU unit has been developed employing memory re-use and parallel computations to obtain considerable speed-up of 26x in comparison to the software baseline. The code and video presentation for this project are made available at the end of report.

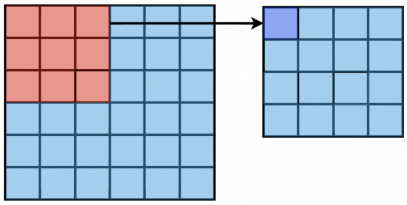


Figure 1: Illustration of single channel 2D convolution using a 3x3 filter

2 Review of existing frameworks

There have been significant number of frameworks relevant to hardware acceleration in the recent times. The review is primarily focused on frameworks that are more recent and relevant to the case of CFU playground. One of the frameworks, hls4ml[2], by Fahim et al., translates models from frameworks like PyTorch, Keras to a HLS project, in addition to providing in-built methods for pruning, quantisation and data-representations of intermediate weights, making it end-to-end. The framework converts each layer to a separate configurable module based on existing templates (uses streaming architecture for convolution)[3] and these blocks could be cascaded and configured based on user constraints for latency, throughput, power and resource usage. For example, the user inputs a re-use factor R for the DSP slices, based on which the degree of parallelism is determined. Once the HLS project is configured, Xilinx tool-chains can be used to generate RTL code for the IP. A similar framework, autoDNNchip[4] by Xu et al., provides multiple Pareto optimal choices for a given set of constraints, and is mainly based on two modules- chip builder and chip predictor, which operate on a graph based representation of the DNNs using template IP architectures. While the chip predictor performs analytical estimations to rule out infeasible designs, the chip builder performs deep inter-IP pipelines and further optimisations to give a set of optimal candidates. Both of these frameworks generate the RTL code for the whole network instead of specific bottlenecks which are common to several networks, which makes them difficult to re-use for a network different from the one configured. Further, the IP is designed in isolation from the core and it could so happen that when integrated the memory transfer might lead to a bottleneck. Taking this into consideration, frameworks like Gemmini[5] by Genc et al., adopt a full-stack approach with a complete SoC integration with the host CPU. To address the hardware flexibility issue, Gemmini uses a two-level hierarchy for MAC units. They constitute a spatial array of tiles separated by pipeline registers, and a set of PEs in the tiles connected combinationally- this enables both systolic array implementation using the pipeline registers, as well as parallel vector computations using combinationally connected PEs. The user can also co-design custom virtual address translation schemes, partition system-level memory resources like cache, RAM etc., while the whole operation is still performed on the accelerator similar to the previous cases. The framework CFU-Playground which is being used for this project, has many features similar to Gemmini in terms of full SoC integration of CFU and co-design schemes. This framework additionally provides specialised instructions which shall be executed on the accelerator in the case of specific bottlenecks, still executing a considerable amount of code on the CPU unlike other frameworks like hls4ml accelerating the whole operation on Hardware, where CPU plays the role of sending the initial inputs only. The same has been detailed in the following section.

3 CFU-Playground

The framework CFU-Playground is completely open-source and is based on open-source tools like LiteX, VexRiscv, nMigen, TFLite etc. An integrated framework with complete SoC integration is adopted to avoid end-to-end storage and network bottlenecks when the host CPU is integrated with the accelerator. Unlike the previous frameworks, the user needs to design the accelerator in this case, and CFU-playground only offers a ground for the integration of accelerator. The Custom Function Unit, similar to the ALU is a light-weight accelerator that could be connected to the VexRiscv CPU core. This allows the model-specific acceleration while at the same time performing a significant amount of computation on the CPU by speeding up only the required "hot-spots" on the lightweight CFU and leaving the execution of outer loops to software. Methods like parallel computation of independent operations, pipelined execution as well as storing activations for re-use shall be possible design explorations for speed-up, while at the same time the frameworks also supports configuring the VexRiscv CPU core for pipeline stages, caches, FP units, which allows to tune the CPU based on the CFU. The SoC design and the placement of CFU has been re-drawn from source and presented in Fig-2. Clearly, from the placing of the CFU, it has direct access to registers of the VexRiscv core, unlike most accelerators which communicate with the core using interfaces like AXI - thus, when synthesised this prevents storage and network bottlenecks post integration with the core. Further possible improvements could be direct memory accesses to the CFU, which enables streaming of contiguous blocks of data which is the case for most applications.

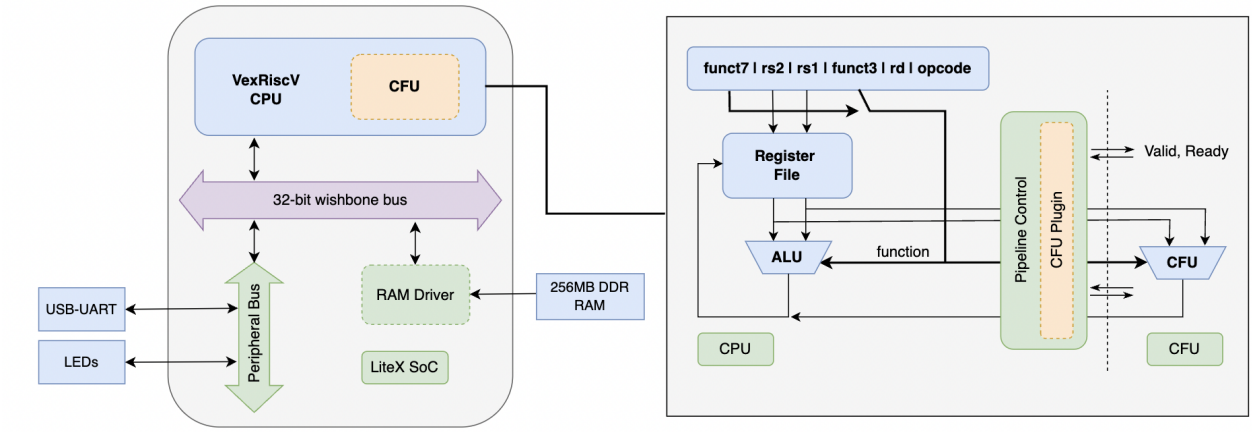


Figure 2: LiteX SoC Integration and the data-flow of CFU, where the CFU is controlled and operated using the R-type instructions of RISC-V

The CFU is invoked using custom instructions in the C-code, and these correspond to the R-type instructions in RISC-V assembly. To invoke the CFU, macros like "q = cfu_op(func7, func3, a, b);" are provided which expand to “asm” directives when compiled, and these can be used by the user in the high-level C code to call the CFU. Consider the below lines of code being used in general C code -

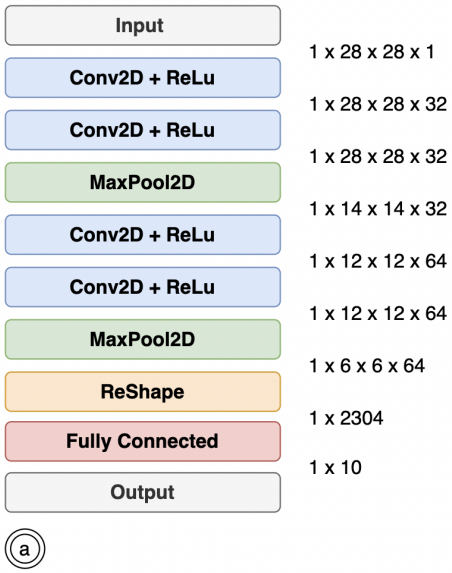
```
a = 5;
b = 6;
c = cfu_op(0, 0, a, b);
```

The compiler translates the above macro to an R-type instruction. It sets the registers where a, b are stored as well as where the result is to be stored and using the func3 and func7 provided by the user the translation is performed. The handshaking is performed using ready and valid signals. The CFU is activated by the CPU using a valid signal and the contents of a, b, func3, func7 are passed as an input to the CFU, which operates on the inputs and gives the result, which is stored back at the destination register. The framework allows the usage of macros in TFLite kernels like Conv2D. Since inference of any Neural Network contains multiple instances of Conv2D kernels, using a CFU in these allows the same kernel to be re-used for multiple Neural Networks, without changing the underlying hardware. To build the accelerator itself, the user could either use Verilog or the framework offers the support for nMigen, which converts accelerators written in Python to RTL. The built accelerator could be tested using most of the FPGA boards supported by LiteX which provide accurate cycle counts. To verify the functional correctness of the accelerator, the user could either use Verilator or Renode Simulation. Although Renode is not cycle accurate, this could help to check the functionality. In practise, between successive calls of the CFU there could be junk cycles where the input data to the CFU could lead to errors if all the control signals used aren’t masked by the valid signal provided to the CFU. The Renode doesn’t handle this case as it doesn’t simulate those junk cycles, and it could be possible that the accelerator is faulty and still passes all the functional tests under Renode simulation. This was verified using traces generated with both Renode and Verilator. Hence, it is beneficial to test the accelerator using Verilator, although it is slower than Renode.

4 Software Baseline

A simple multi-layer Neural Network was trained on MNIST using TensorFlow and the same has been quantised to int8, since the fixed-point representations provide less complexity in terms of hardware in comparison to the floating-point representations. The validation accuracy of the float model is 0.9915, while that of the quantised model is 0.9912, indicating negligible loss of accuracy post quantisation. The network architecture used has been presented in Fig-3a. The network operates on a single channel gray-scale [28 x 28] image and the final result is the prediction score of each class. The intermediate layers have significant number of channels, making the network large enough. The usage of int8 representations allows to transfer 4 elements at a time on a 32-bit bus and the same shall be of use when transferring data to the CFU. A theoretical estimate of the number of Multiply and Accumulate operations required by each layer has been presented in Fig-3b. The cycle counts obtained on inference using a Nexys4 Artix-7 board have also been presented in Fig-3b. Clearly the "3x3" Conv2D operation consumes significant amount of time in comparison to that of other operations. Further, the number of cycles required for the inner-most loop of a convolution operation involving a MAC operation were also obtained by instrumenting the

base code using the performance counters of the CPU core. Software optimisations like unrolling on the input-channel dimension have been performed on the Conv2D kernel in order to reduce the branch-overhead. The framework also suggests to replace model specific parameters with constants to reduce memory overheads, but in-view of re-usability of the kernel for multiple networks the same was not performed.



Layer	kMACS
Conv2D-1	225
Conv2D-2	7225
Conv2D-3	2654
Conv2D-4	5308

Layer	Million Cycles
Conv2D	19.682
Conv2D	153.151
MaxPool2D	0.76
Conv2D	59.126
Conv2D	101.266
MaxPool2D	0.278
Reshape	0.027
FC Layer	0.32
Total	~ 335

b

Figure 3: a) The architecture of the convolution neural network used for the development of the accelerator and profiling the baseline. b) Top: Theoretical estimate of the number of MAC operations per each convectional layer in the Network. Bottom: Cycle counts obtained on Nexys4 Artix-7 FPGA for the inference of baseline TFLite Kernel.

4.1 Analysis of Baseline

The total number of cycles taken for the execution of Conv2D operation is nearly 334 million cycles, which asserts the previous argument that the convolution operation takes the most time for execution in the case of above Neural Network. Within the convolution operation the MAC operations (inclusive of reading the data and performing multiply accumulate) take nearly 310 million cycles, which enforces the claim of MAC operations being the "hot-spots" in the case of Conv2D. The ratio of total cycles counts $(2.6 : 1 : 1.7)$ between layers 2, 3, 4 are approximately the same as that of the ratio of estimated number of MACs $(2.7 : 1 : 2)$ which further supports the view of MACs playing a dominant role on the cycle counts. While this is the case, although the ratio of MACs for layers 1, 2 is 32 but the layer 1 only runs nearly 8x faster than that of layer 2. This is because each of the layers 1, 2 have to write back the same number of activations and further since the inputs to the initial layer are the images itself being read from main memory and not the activations which might end-up in cache when stored by the previous layer. Thus, on a whole the MACs form the bottleneck not only in terms of compute, but these also involve equivalent number of accesses to memory, some of which might end up in the registers or cache or can go up to the main memory. Further, unrolling the code along the input depth dimension reduced the cycle counts to 220 M cycles and the speed-up could be attributed to the reduction in branch overhead and better utilisation of spatial locality of the cache, when compared to the previous case.

To tackle this bottleneck, there could be multiple approaches for parallelism while at the same time having efficient data-flow for memory re-use. There are several implementations in the literature on optimising memory accesses for convolution computation by varying the data flow along with the usage of on-chip buffers.[6,7] While these are developed for discrete accelerators performing the whole operation on hardware, the general principles still hold true. On observation, the activations across the input depth could be computed and accumulated in an SIMD fashion. Further independent strides could also be computed parallely having separate accumulator for two elements in the output channel. While the these cases aim at paralleling computations, it is necessary to reduce the memory access to gain from parallelism. Some forms of parallelism like broadcasting the input data shall inherently reduce the memory accesses as well. Further, there exists elements in common between the successive strides and computation of multiple output channels depending on the same input channel, both of which gain from storing the input activation to buffers for re-use of data. Based on these inferences an accelerator CFU was developed, and the software was unrolled and modified to feed the CFU appropriately and process the outputs.

5 CFU Hardware Accelerator

Based on previous inferences, an accelerator equipped with a small scratchpad memory for storing the input activations, PEs which compute the output activations as $(4, 4)$ tiles and an accumulator buffer for swapping the output tiles for input activation re-use is developed. Although the size of the accumulator buffer is fixed, re-use can be controlled in software using a parameter L, by keeping the input activations fixed, streaming the weights corresponding to each of the channels and swapping the current accumulator from the buffer appropriately between the channels. Since the CFU offers 32-bit reads and writes like the general ALU, we could send eight int8 activations using two registers simultaneously to the CFU in contrast to outputs read separately since they are int32 representations. For an input channel of dimension D, for say, sixteen, a $(4, 4)$ tile of the output depends on $(6, 6, 16)$ input tile, which is convoluted with $(3, 3, 16)$ weight tile. The accelerator has sixteen PEs, each of which computes one element corresponding to the $(4, 4)$ tile of the output. To change the mapping of the output channel of a tile to the PE array, each PE has an accumulator buffer and to select an element from the buffer, the input signals from the CPU are used. The data-flow for a simple case has been illustrated in Fig-4, along with the PE architecture. The following case provides a sketch of the general operation of the CFU. The $(6, 6, 16)$ input tile could be broken down to four $(6, 6, 4)$

sub-tiles each indexed as $(0-3)$. We first store the 0th input tile in the scratchpad memory of the accelerator. Of these input elements, we store two elements of dimension $(1,1,4)$ at a time and the memory address pointer of the scratchpad increments automatically by two words after each write (reset to 0 before writing the whole chunk). Thus, the write macro must be invoked 18 times to store the complete input tile. Once the input activation writing is completed, we accumulate to a $(4,4)$ tile of the PEs by streaming the weights. Suppose the accumulator buffer is being used till a length of L; we could use this input on the scratchpad memory to compute the output of L channels, which requires the streaming of L, $(3,3,4)$ weight tiles corresponding to L channels of the output, with swapping the accumulator buffer between successive channels. We first load and accumulate $(3,3,4)$ tile of the weight corresponding to the first of L channels. In the PEs, the first accumulator is selected, and on each invocation of the CFU, the $(1,1,4)$ weight input vector is broadcasted to all the PEs. The input corresponding to each PEs from the input scratchpad is fed. The computation is performed in a SIMD fashion along the input depth dimension, as represented in Fig-4. After accumulating nine $(1,1,4)$ weight elements of the first channel, the second accumulator is selected, and the $(3,3,4)$ weight tile of the second channel is streamed. This way, we continue for L channels of the output. Then we load the next $(6,6,4)(1)$ tile of the input and perform the same set of operations till we reach $(6,6,4)(3)$. More generally, we traverse the whole input depth D. At the end of the process, the output's whole $(4,4,L)$ tile is accumulated and stored back in the activation memory.

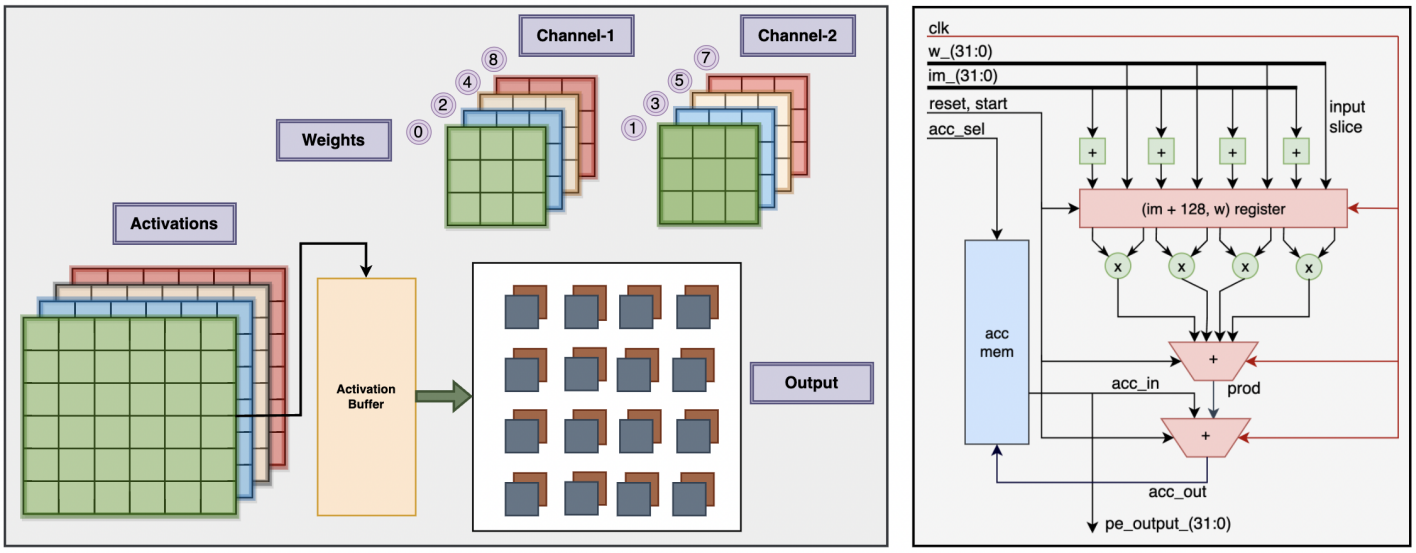


Figure 4: Left: Represents the data-flow of proposed accelerator, each of the coloured input activations and filter weights are of depth 4. L is considered as 2 for this case. Intially the green map is loaded to the scratchpad memory. Weights represented by 0 (channel-1) are streamed and the PEs store the accumulated values to the their initial location of the buffer. Then weights represented by 1 (channel-2) are loaded, and the PEs accumulate the result to next location on the buffer. Next activations represented by blue are loaded to perform the same set of operations. The process continues till the input depth is traversed. Right: The PE architecture used in the accelerator with SIMD computation along the input depth. It is pipelined for two stages in order to reduce the influence on the critical path. The location on the accelerator buffer is selected based on the inputs to the CFU.

In the architecture built using Verilog HDL, the accumulator memory could store 8, 32-bit values, which corresponds to 512 bytes for all the PEs, while the input scratchpad is chosen to be 256 bytes making it a total of 768 bytes of Memory on the CFU. Further, since the amount of memory is considerably low, registers could be used to implement the same enabling low-latency data reads. The PE has been pipelined for two stages to cut down the critical path, which means the result is only accumulated at the desired location after a two cycle delay. The same is taken care of in software by adding flush cycles which shall feed zeros to the PE, so that no discrepancies arise while shifting the accumulator to change the mapping of channels. Further, the inputs to the CFU itself are pipelined which means that it could accept data on consecutive cycles. The detailed schematic of the accelerator is presented in Fig-5, along with the input control signals provided by the CPU. In the software, the Conv2D kernel was modified appropriately by unrolling the loops iterating on input depth, output width, output height by a factor of 4, while the loop iterating on output depth was unrolled by a factor of L (can go upto 8) which controls the re-use of input data as inferred from the previous case. The loops are arranged such that output activation elements are computed in strides of $(4,4,L)$. The performance by varying the value of L has been analysed, along with modifying other CPU related parameters, to obtain optimal speed-up. Further the kernel was tested for generality when used for other networks by using a larger version of the current network. The whole design was integrated with the VexRiscv CPU core and the same was synthesised, placed and routed without any errors on Nexys4 Artix-7 FPGA using Xilinx toolchain. The same was tested for functional correctness using the tests ran from the software stack on multiple input images. The Xilinx reports are made available in the code repository.

5.1 Analysis of Integrated Design

The accelerator was synthesised, placed and routed on the same FPGA as that of the baseline. To check the effect of CFU on the critical path, the PE module was iteratively pipelined for two stages which helped in the reduction of the critical path. Further, the CFU could accept the inputs on consecutive cycles and the data to the PEs and scratchpad memory is provided synchronously. Both the integrated accelerator and the CPU alone were synthesised for a 100 MHz clock on the FPGA with a worst negative slack of 0.554 ns and 1.215 ns respectively. From the WNS, the critical paths estimated are 9.446 ns and 8.785 ns respectively, indicating a minor decrement in the critical path after the addition of CFU in comparision to the actual clock rate. For analysis, as the decrement is minor, we assume both the designs are running at same clock and that is the case on the Nexys4 Artix-7 board, and the cycle counts shall provide the absolute speed-up. This also enforces the advantage of integrated SoC environment provided by the framework over being designed in isolation without considering the end-to-end bottlenecks. Further, the resource consumption was completely in the limits of the FPGA board.

The total number of accesses to input activation elements by all the layers in software baseline is same as that of the number of MACs and is given by 15412 K. For $L=1$, in the method of computing the outputs always as $(4,4)$ tiles, the total number of accesses made

are given by 3839 K (assuming int8 reads), which is approximately 4 times lower than that of the previous case. Here the reduction of memory accesses was due to the re-use of activation data between the successive strides of convolution limited to $(6, 6)$ tile in the input activation map. The total number of access made to the weight array are given by 963 K, which is approximately 16 times lower than the previous case of baseline, and the decrease is due to broadcasting of weights to 16 PEs on one read. Along with these, the parallel computation of $(4, 4)$ tile outputs with the SIMD accumulation of elements on the input depth dimension, caused a reduction in number of cycles to 29 M from the 335 M of the baseline. While this is the case, not only there shall be a reduction in number of accesses made to input activation, this shall also provide better spatiality since inputs are being read as 32-bit chunk along the input depth dimension, and this also adds to speed-up. The code was profiled to infer the time spent at each of the sub-parts. A total of 17 M cycles was spent at sending the input data to the scratchpad, 5.5 M spent post-processing and writing back to the output activations, 4.6 M sending the weights to the CFU and performing the accumulations. The time spent in sending the input activations is significant in comparison to streaming weights to accumulators, and this could be because, the reduction in number of memory accesses is less in comparison to that of weights and since the weight array being small in dimension has the advantage that the elements might remain in cache even after a longer interval. This also justifies the data reading might have been the bottleneck while performing MAC operations in baseline. The decrease in the number of cycles spent from 13 M to 5.5 M in post-processing and writing back can be attributed to better spatiality of cache where multiple elements are being written around the same time. Further, the massive unrolling of several loops also adds to the reduction in the branch overheads as the number for loop executions reduced by 4 times on the output-width, output-depth and input-depth dimensions, while they are completely eliminated on the filter-width and filter-depth dimensions.

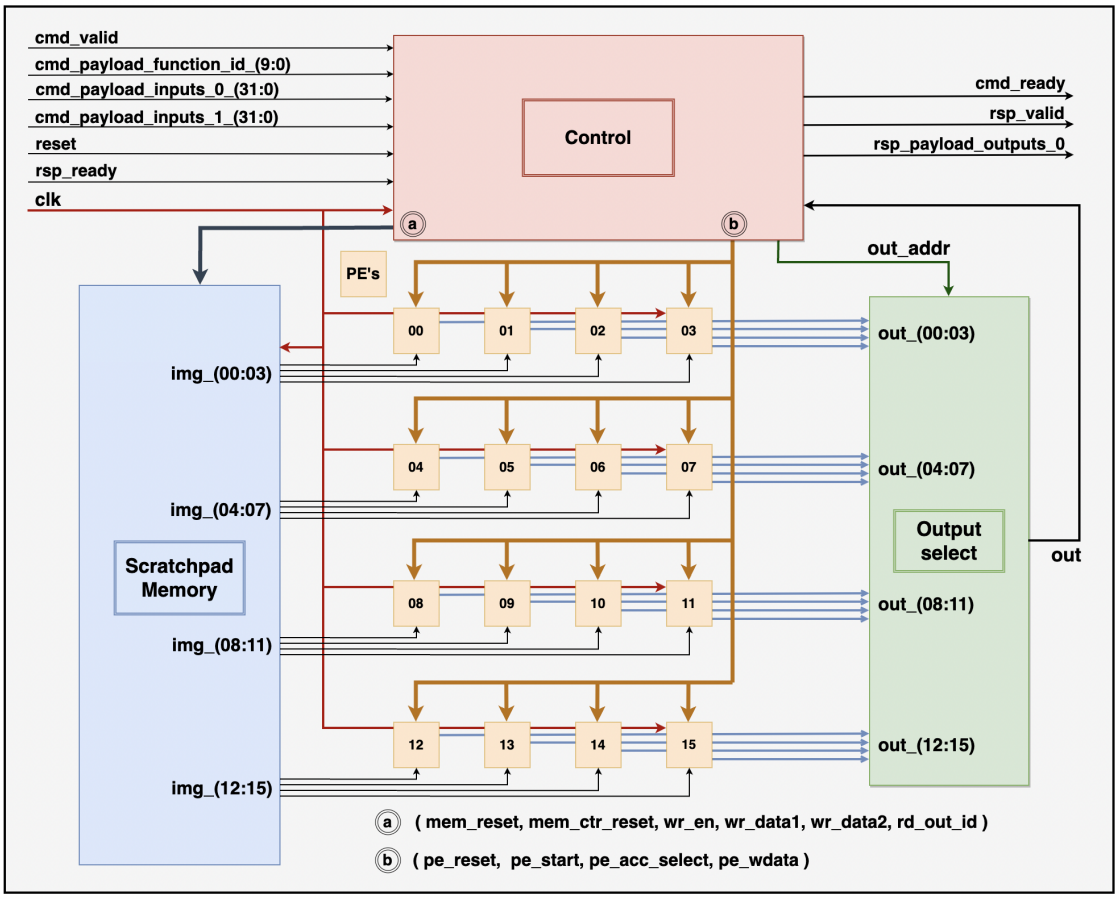


Figure 5: The architecture schematic of the CFU module built in Verilog. Two internal counters are maintained one used for generating the write address of the input scratchpad memory, while the other used to feed appropriate inputs to the PEs for accumulation. The function-id is used to control the operations performed. The func (sliced from function-id) value 4 is used to set these counter to zero. Further, the accumulator buffer could be set to zero using value 3. The value 5 is used to perform writes to the input scratchpad and the counter increments automatically. The value 1 is used to send the input filter values and perform the accumulations. The output could be read using the func value 2. The CFU module is pipelined and can accept inputs in consecutive cycles.

Based on the previous inferences, reducing the memory accesses to input can help to reduce the cycle count, and this could be done by re-using the elements in input scratch-pad memory. The value of L was changed first to 4 and then to 8. Although it could be increased further, the accelerator supports till 8, so as to make it light-weight while at the same time providing significant amount of speed-up. As we move to $L = 4, 8$ the total cycles expensed are 20 M and 15 M and the time spent at the writing the inputs to scratchpad reduced to 5.2 M and 3.3 M respectively which corresponds to a decrease of 0.7 and 0.8 times, and this holds on comparision to the previous case where as L increases the accesses reduce by a factor of L. The reduction in the number of cycles could be mainly attributed to the re-use of input activations rather than loading them from the memory. While this is the case the number of accesses to weight values remains the same and there wasn't significant change in number of cycles for different values of L. There is a slight increase of output processing cycles and this might be caused because of increase in branch overhead by addition of an extra loop iterating on L to store the outputs. Further, the length of the accumulator buffer could have been increased in-order to increase the value of L,and the cycles may reduce by a factor of L. However, after a certain stage, the decrement in the number of cycles is governed by Amdahl's law and may also lead to excess usage of the resources.

The accelerator code was tested for changes in the cache structure and decreasing the number of bytes per line keeping the size of the cache same. Using 8 bytes per line, the cycle count was reduced to as low as 13 M and the clock being same as the previous case the WNS obtained was 0.662 ns, indicating a better critical path of 9.338 ns. The decrease in cycles might be due to the reduce in cache line conflicts by the reduction of bytes per line and the increase in the cache locations. The final CFU was also tested with a different large network with double the number of channels in comparision to the current network and the inference was completed in 38 M cycles, while the baseline took 1268 M cycles, and this suggests a massive reduction in cycles due to better re-use of memory while at the same time performing parallel computations. Thus, the overall speed-up obtained for the original network was nearly 26x while that for the large network was 33x.

There is a scope of further optimisation. On inspecting the trace generated by Verilator, it was found that the weight streaming and accumulation was not being done every consecutive cycles rather there was some minor latency involved in sending of the weights. Since weights are something fixed to a given network when an application is deployed, the inference cycle counts could be potentially reduced further by initially streaming all the weight data to the accelerator. Post this, the inputs could be sent to scratchpad, and the loop counter in hardware could be used to perform the accumulation.

6 Results and Conclusions

- The framework CFU-Playground was reviewed in contrast to other existing frameworks, and particularly, its advantages like Integrated SoC environment, significant usage of the CPU core apart from the initial data transfer etc., have been exploited in this work.
- The Conv2D operation for a 3x3 case in an MNIST Neural Network was analysed and the Software Baseline was set-up in order to check for the bottlenecks. The software baseline for the given network took 335 M cycles for execution on a VexRiscV core placed, routed on a Nexys4 Artix-7 FPGA. The Conv2D operations consumed 334 M cycles of the total cycle count, and within the Conv2D operation the MAC operations were the bottleneck being executed for 310 M cycles. The code was unrolled to reduce the loop overheads and gain from the spaciality of cache, which enhanced the cycle count to 220 M cycles.
- Using methods for parallelism like SIMD accumulation along input depth, parallel computation of independent strides and input data re-use between strides as well as multiple output channels, an accelerator was built. The integrated core when synthesised had a critical path of 9.446 ns in comparision to 8.785 ns for the core, indicating a minimal increase owing to the Integrated SoC environment. The inference took 15 M cycles on this integrated core.
- The cache of core was slightly modified to lower number of bytes per line and on synthesis the the critical path was enhanced to 9.338 ns. The network took 13 M cycles to execute on this integrated core.
- The overall speed-up obtained was 26x for the base network and when tested for kernel re-use on a larger network the speed-up obtained was 33x. This was using the assumption that the baseline and the integrated accelerator were running at same clock since the difference in critical paths is minimal. Thus, the framework provides a better environment for the development of accelerators and the same was used to achieve the acceleration of 3x3 Conv2D kernels in case of a MNIST Neural Network, with a significant reduction in cycles.

7 Code & Demo Video

The repository with the code for this project could be found at `https://github.com/sgauthamr2001/Conv2D_CFU`, and the video presentation for the project could be found at `https://bit.ly/conv-cfu`.

8 References

The following items have had a significant contribution in shaping up this project. In addition to these, several other papers and articles have been reviewed.

1 : CFU Playground: Framework for Tiny Machine Learning (tinyML) Acceleration on FPGAs, Prakash et al.

2 : hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices, Fahim et al.

3 : SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures, Alexander et al.

4 : AutoDNNchip: An Automated DNN Chip Predictor and Builder for Both FPGAs and ASICs, Xu et al.

5 : Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration, Genc et al.

6 : MEM-OPT: A Scheduling and Data Re-Use System to Optimize On-Chip Memory Usage for CNNs, Dinelli et al.

7 : CARLA: A Convolution Accelerator with a Reconfigurable and Low-Energy Architecture, Mehdi et al.