Using HLS to Accelerate MAERI - Final Report

Anirudh Itagi
Georgia Institute of Technology
Atlanta, USA
aitagi7@gatech.edu

Hanjiang Wu
Georgia Institute of Technology
Atlanta, USA
hwu419@gatech.edu

Geonhwa Jeong
Georgia Institute of Technology
Atlanta, USA
geonhwa.jeong@gatech.edu

I. Introduction

Deep Learning (DL) has been extensively influencing our daily lives in personal recommendations, autonomous vehicles, language translation, etc often exceeding the performance of humans. To further enhance the operational performance of DL models, researchers have been working on making those models larger and deeper with billions of parameters. The increased number of parameters causes an enormous number of computations, inducing enormous power consumption as well as a long training time and inference latency. To address the issue, various specialized accelerators have been introduced to reduce the computational burden from both industry and academia, including, but not limited to, Google TPU [4], Graphcore's IPU [3], Eyeriss [2], and MAERI [8] using a large number of parallel compute units with efficient interconnection networks to distribute workloads to the compute units and reduce the partial results from the compute units.

The specialized accelerators often adopt specific dataflow that is fully customized for the target workload leveraging the prior knowledge of the target workload to maximize the data reuse for reducing memory traffic as well as saving power. Even though the specialized accelerators target some specific workloads (DL model for DL accelerators), there are two main challenges for accelerator developers. First, there are many operations that should be supported in DL models, which might require different optimizations or dataflows. For example, most of the accelerators only focus on convolutional layers or fully connected layers, but once they get accelerated, it is possible that other layers could become new bottlenecks. Next, even for a single operation, such as a Convolution operation, different dimension sizes could be used, such as different filter sizes or different numbers of channels in each kernel, which would prefer different dataflows to fully utilize the underlying hardware resources. Previous works [6], [9], [10] have shown that there is no single universal dataflow that works efficiently on operations with different dimension sizes.

MAERI [8] is a flexible accelerator that can support different dataflow to be efficient for different dimension sizes. However, the original version was only focusing on Convolution operation and the authors only provide RTL code instead of an end-to-end framework. Thus, MAERI 2.0 is under development to provide end-to-end framework as well as support for various operations, using HLS. In this work, we will be working on adding more features on MAERI 2.0.

II. Background

As mentioned before, many accelerators have been introduced recently, but we focus on MAERI [8] as it is the most relevant one for our project in terms of flexibility.

A. MAERI

With the increasing complexity and variability among different DNN models (Convolution, LSTM, fully-connected), reconfigurability becomes one of the crucial design aspects. Unlike previous accelerators that do no support full flexibility, such as Eyeriss [2], MAERI has flexible interconnects to facilitate different dataflows, so it can extend the implementation of row stationary to output stationary depending on the given workload.

Another prominent characteristic of MAERI is the design choice of the Augmented Reduction Tree (ART). Compared to the existing work with flexible communication that uses all-to-all meshes as the Network on Chip [7], ART has significantly less power consumption and area utilization. More importantly, the proposed ART structure is a superset of the most popular DNN accelerators that most of them utilize a tree-like structure, so MAERI is able to implement different type of workloads without the without loss of generality.

The flexible interconnects are realized by the reconfigurable switches that can be tuned to support different data mappings. There are three types of switches in MAERI including simple switches, multiplier switches and adder switches. With the help of a programmable controller, those switches facilitate the data movement between tree layers and achieve the goal of supporting different dataflows.

III. Problem Description

The inference process of a ML model involves the implementation of various dataflows, tiling, ordering, parallelism, and shapes. Applying an appropriate dataflow can significantly minimize the latency of the inference process by up to two orders of magnitude compared to an inefficient one [5]. This highlights the importance of selecting an optimal dataflow for ML model inference to achieve enhanced performance.

The process of adapting hardware to accommodate varying dataflows requires on-chip reordering of data and reconfiguring of datapaths, which introduces significant overheads. These overheads can restrict the ability of ML accelerators to utilize diverse dataflows, leading to suboptimal performance. Thus, the challenge lies in mitigating the overheads

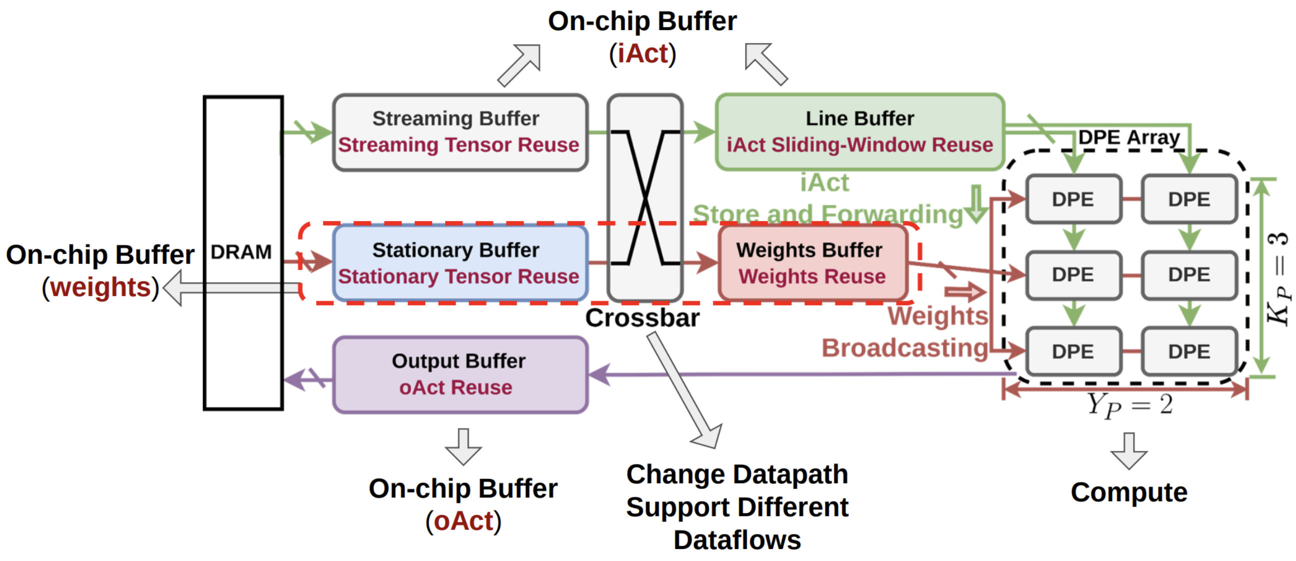


Fig. 1. MAERI 2.0's Microarchitecture implemented on the ZCU104 FPGA [1].

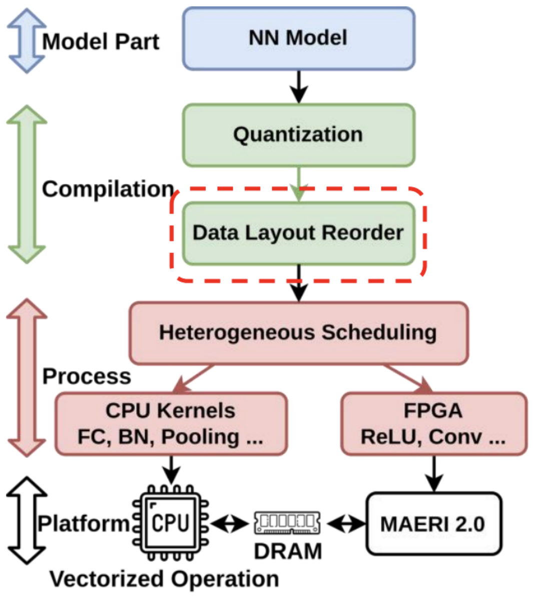


Fig. 2. MAERI 2.0 End2End inference framework [1].

associated with re-configuring hardware to facilitate efficient utilization of different dataflows by ML accelerators, thereby achieving optimal performance.

IV. Main Work

In this section, we show our main contributions for this project, including Arbitrary IFC-Buffer burst transfer, ReLU and Batch Normalization, and max pooling.

A. *Arbitrary IFC-Buffer Burst Transfer*

The need for this implementation arises due to a disproportionate ratio between AXI interface ports and on-chip (PL) buffer (BRAM) partitioning, that supports PS-PL data transfer, with the additional constraint to ensure burst transfer while ensuring optimal on-chip memory utilization. Take for instance the scenario when the on-chip buffer is partitioned by a factor of 4 and the number of AXI ports available are 6. In this scenario it is trivial and inefficient to increase the partitioning to 6 or decrease the number of AXI ports used to 4, because as a consequence, either the buffer is over-allocated and underutilized or the data transfer speed is reduced due to some AXI ports being underutilized.

In order to solve this issue, we propose a software and hardware co-design solution that reorders data that is intended to be stored into the partitioned buffer and fills the buffer in a wrap-around fashion while meeting the above mentioned constraints. In the following paragraph, we discuss how our wrap-around algorithm works to engender burst-mode transfer.

Figure 3 shows a simple data transfer diagram of 4 cycles. Each vertical green blocked lane represents an AXI interconnect, and the stars with the same color express the data coming from different ports (6 ports in total as mentioned above). In the Cycle-1, we have the upper-left spot filled by a red star coming from the 1st port and with an interval of 17 empty spots, we put in the next data point coming from the 2nd port. The following red stars coming from 3rd-6th ports are interleaved in the same fashion of 17 data spots apart. Here, the value 17 is deduced from the formula: 4n + 1, with n being the spots separated by two consecutive data points. In Cycle-2 of the same Figure 3, we insert the data on 4 AXIs with position one right-shifted relative to Cycle-1. From the data placements in Cycle-1 and Cycle-2, we can infer that

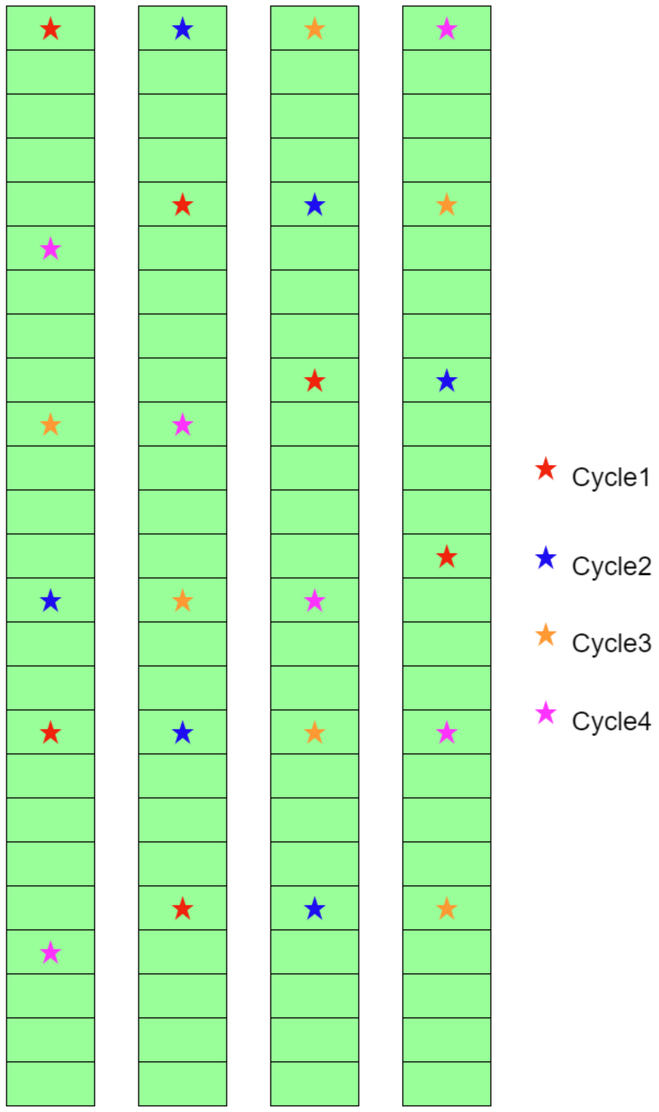
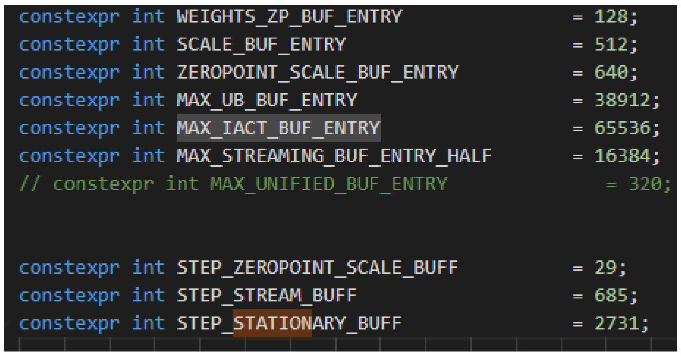


Fig. 3. Diagram for the flow of burst-mode transfer.



```
constexpr int WEIGHTS_ZP_BUF_ENTRY              = 128;
constexpr int SCALE_BUF_ENTRY                   = 512;
constexpr int ZEROPOINT_SCALE_BUF_ENTRY         = 640;
constexpr int MAX_UB_BUF_ENTRY                  = 38912;
constexpr int MAX_IACT_BUF_ENTRY                = 65536;
constexpr int MAX_STREAMING_BUF_ENTRY_HALF      = 16384;
// constexpr int MAX_UNIFIED_BUF_ENTRY             = 320;


constexpr int STEP_ZEROPOINT_SCALE_BUFF         = 29;
constexpr int STEP_STREAM_BUFF                  = 685;
constexpr int STEP_STATIONARY_BUFF              = 2731;
```

Fig. 4. list of parameters for the burst-mode transfer.

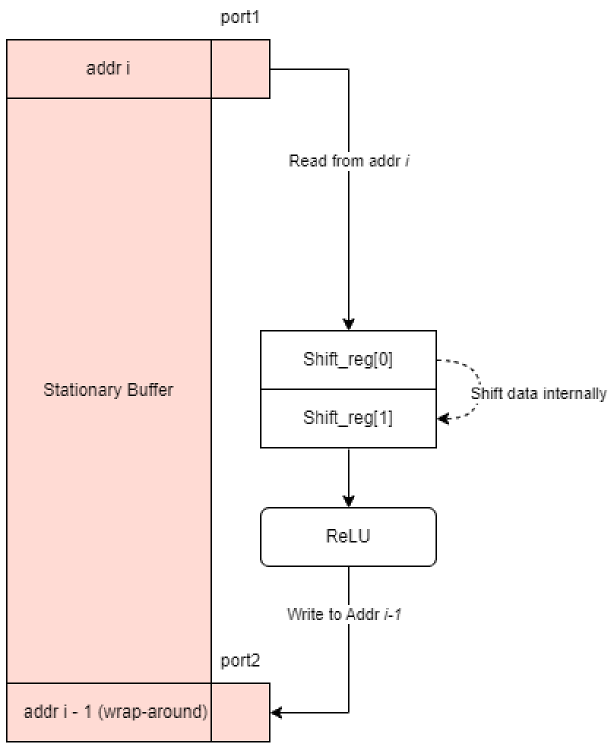


Fig. 5. Internal Shift Register for ReLU and Batch Normalization.

each port puts data next to each other, so this implies that 4n of slots on the buffer are reserved for the data coming from one port. The reason behind the interval between two data moved to the buffer being 4n+1 is our cyclic and wrap-around algorithm. The 4n will end up with the next data put on the same AXI and sent to the same sliced buffer. On the FPGA, there are only two ports for each of the partitioned buffers, so we cannot put all the data on the same AXI. More importantly, we want to maximally utilize all 4 AXIs in one burst and resulting in the formula of 4n+1.

In the Figure 4, we displayed a list of parameters that we are using for the demonstration and all of those are real parameters in our implemetation. MAX_IACT_BUF_ENTRY is the stationary buffer budget for storing the input activaions which is based the largest convoluted layer we would encounter in the market. Given 6 ports at the DRAM edge, we divide the MAX_IACT_BUF_ENTRY by 6 and round that value up to the larger integer, ceil(65536 / 6) = 10923. Then, we use the formula of 4n + 1 to get the number of step for the step interval between two consecutive buffer placements in a single cycle. This will get us n = ceil((10923 - 1) / 4). To make sure our wrap around works well among the four AXIs, we still apply the ceiling fuction on the results of division. At the end, we will get the value of n = 2731. This is how we get the STEP_STATIONARY_BUFF = 2731 as shown in Fig 4. The same logic is applied to the data transfer for all the on-chip buffers. By doing the burst-mode tranfer, we are able to get an approximately 3x speedup compared to the original MAERI implementation.

B. Batch Normalization

Following the Pytorch implementation, the Batch Normalization has the following equation, in **??**. In MAERI2.0, we reduce the equation to y = mx+b, referring to **??**. In this work, we focus on the inference part of the workloads, so we assume all the coefficients are constants as they are fixed during the training. Also, the values of m and b are quantized in 8bits, on the same scale as zero point. In the following, we show how we calculate y_{8bit} from x_{8bit}.

1) $x_{8bit} -> x_{fp}$
   First, we need to convert 8 bit quantized value to fp value.
2) $x_{fp} = (x_{8bit} - ZeroPoint_{8bit}) * Scale_{fp}$
   The conversion is done using Scale and ZeroPoint constants.
3) $y_{fp} = \gamma * (\frac{x_{fp} - E[x]}{\sqrt{Var[x] + \epsilon}}) + \beta$
   Here, we show the equation for the batch normalization.
4) $y_{fp} = M_{fp} * x_{fp} + C_{fp}$
   a) $M_{fp} = \frac{\gamma}{\sqrt{Var[x] + \epsilon}}$
   b) $C_{fp} = \beta - \frac{\gamma * E[x]}{\sqrt{Var[x] + \epsilon}}$

   The above equation can be written as 4) capturing complex terms with M and C. This is important as we would like to pre-calculate as much as possible.
5) $y_{fp} = M_{fp} * scale_{fp} * (x_{8bit} - ZeroPoint_{8bit}) + C_{fp}$
   By substituting x_{fp} with 2), we get 5).
6) $y_{fp} = M_{with\_scale_{fp}} * (x_{8bit} - ZeroPoint_{8bit}) + C_{fp}$
   Again, we can simplify the equation by pre-calculating $M_{fp} * scale_{fp}$ as they are both constants.
7) $y_{8bit} = (y_{fp} * invScale_{fp}) + ZeroPoint_{8bit}$
   Finally, we get the target value, y_{8bit}.

It is noteworthy to mention that the Batch Normalization implementation is compute intensive as it involves data type conversion, 2 multiplication operations, and 2 addition operations per input activation element. However, we would like to minimize the usage of DSPs as they will also be critical for other main operations such as convolution. Thus, to reduce the number of DSPs utilized, we implement our own custom multipliers implemented with LUTs and F/Fs. These are static-configurable designs that will support arbitrary-sized multiplicand and multipliers.

The multiplication is essentially reduced to iterative shift and add. At the moment, we designed three different types of multipliers that use the shift and add technique. The first one is the simplest shift and add implementation without any parallelism. The second implementation introduces 2x parallelism to the computation by splitting the multiplicand two halves, and each level doing the same shift and add implementation as the basic implementation (see Figure 6 for detailed image illustration). For the third and the last one, we divide the data bits into three parts. To demonstrate that we can support arbitrary level of parallelism as well. Further, we have integrated these multipliers into the batch normalization

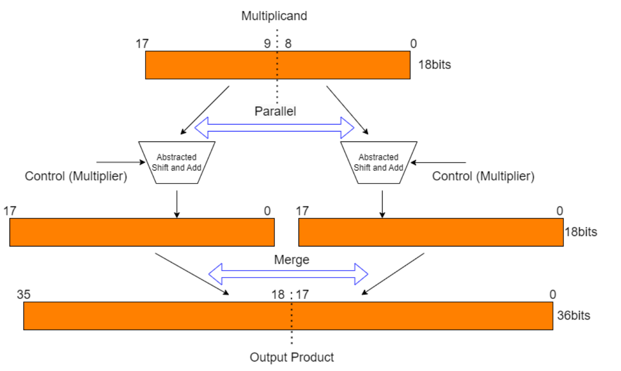


Fig. 6. Two-level parallel shift and add.

operator with configurability in the multiplier implmentation as purely DSP based, purely F/F+LUT based or a hybrid option to spread out the resource utilization.

C. ReLU

The original ReLU was done by sending the input activations from PL back to the PS, which not only suffers from the huge communication cost but also under-utilizes the MAERI's flexible structure. In this new add-on ReLU operator, we keep the input activations within the same stationary buffer. To attain this goal, we have a shift-register that can store two input activation data elements. Every time, we shift the data from the stationary buffer that is for the input activations storage, to the temporary storage of the shift register. The shift register is composed of two slots and each of them is 128 bits which match the performant on-board bandwidth. The following Figure portrays the real implementation of FPGA. The data is first right-shifted to the 0th indexed position of the register and followed by another right shift to the Shift_reg[1]. With both slots being occupied, the data at the 1st slot is then extracted and applied with the ReLU activation function. Here, it is worth mentioning that our ReLU activation is compared with the Pytorch-based quantized zero point under the existing MAERI2.0 framework.The conversion from floating point to fixed point follows the same steps as batch normalization. After applying ReLU, the data is stored back to the same stationary buffer that we get the data from, but in a different location. Say we fetch data at address space i, we will store the data back to address i-1, which eliminates the potential WAR data dependency. In a bigger picture, the whole ReLU work is pipelined, since we have two slots in the shift registers. While we are applying ReLU activation on the data in Shift_reg[1], we are also shifting the data from Shift_reg[0] to Shift_reg[1] and read the data from stationary buffer to Shift_reg[0].

D. RAFFT network and Max Pooling

1) Architecture of RAFFT and its adaptation to Max Pooling: Before talking about the implementation of max pooling, we have to bring up the concept of RAFFT architecture. As an abbreviation for Residual Additive Fat Folding Tree, RAFFT is a customized network that facilitates the max pooling computation. There are two phases in RAFFT, Reduction

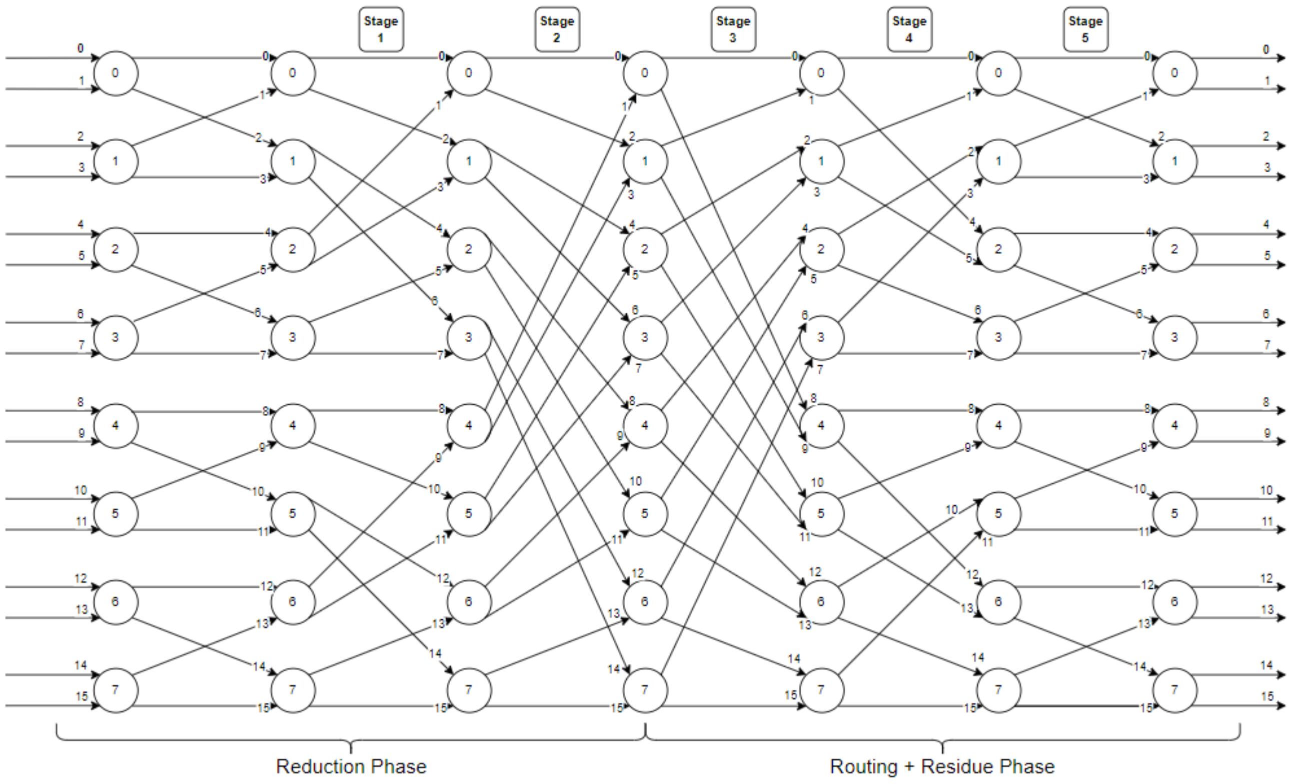


Fig. 7. Structure diagram of RAFFT network.

phase and Routing/Residue phase, as shown in Figure 7. In the reduction phase, we need to reduce all the input pairs through the switches, which are represented by the circled nodes in Figure 7. Each switch takes in two inputs and passes the inputs into a comparator to generate the output results. On the output side, there are two ports and the produced result will come from either one and routed to the next stage depending on the routing algorithm. Algorithm 1 depicts the determination of routing directions inside the RAFFT.

For Max Pooling, there are five basic switch operations. In Figure 8, it exemplifies the flow those five operations inside the switch. The first one is Through, which directly sends the data from the input to the output. For the rest of four operations in Figure 8, we assume the blue colored input comes from the input with 0th index and green colored input comes from the 1st indexed input. So, for the second operation called Switch, the input in green switched the relative position with blue input on the output side. In the Max-through case, both blue and green inputs are passed into the comparator and the result is sent to both output ports. Similarly, Max-1 and Max-0 have both inputs streamed in the comparator and the output can be selected from either 1st or 0th indexed output port.

In the second phase of routing and residue, we take the reduced results (through the comparators inside a switch node) engendered after the first phase and reroute to one of the output nodes on the right most column in Figure 7. The residue under this context means the results generated after

Algorithm 1 RAFFT Routing Pseudo-Algorithm

```
Input: all source-destination pairs, where multiple sources
    could point to the same destinations and the source nodes
    could randomly be scattered.
Output: The configuration bits for AFFT switches.
 1: Sorting the sources-destination pairs based on the number
    of source nodes involved to make sure the larger a group
    is, the higher priority it gets routed.
 2: for all source-destination pairs do
 3:     Find all available middle switches which could poten-
        tially connect all sources and destinations.
 4:     Randomly picks an available middle switch and speci-
        fies the unique paths between all sources and destina-
        tions.
 5:     if non-blocking occurs then
 6:         Return configurations of involved switches.
 7:         Activate reduction for all switches with two inputs
            targeting the same destination.
 8:     else
 9:         Go back to previously routed pairs and select another
            available middle switch for them. Reroute this pair.
10:     end if
11: end for
```

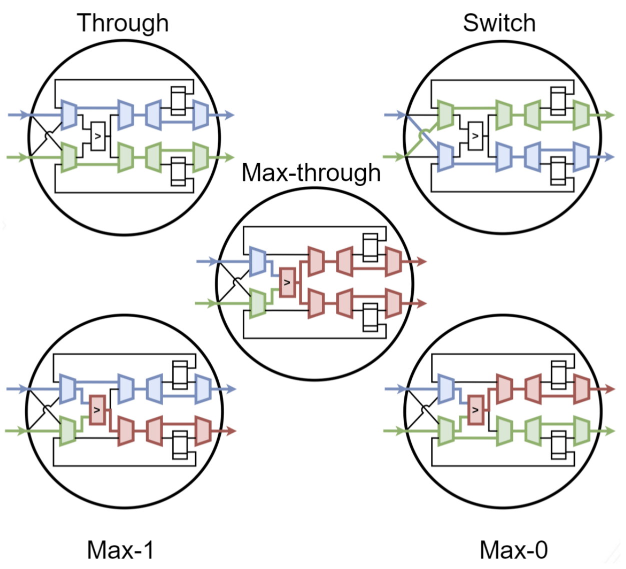
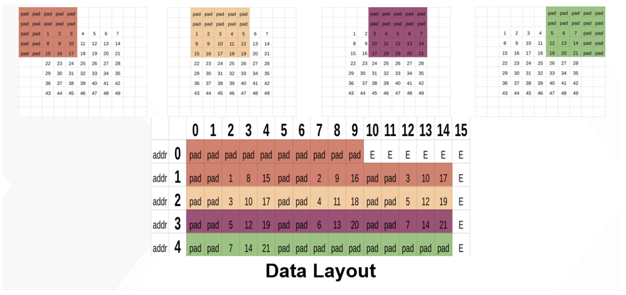


Fig. 8. Five switch operations used for Max Pooling.


Data Layout

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
addr	0	pad	pad	pad	pad	pad	pad	pad	pad	pad	pad	E	E	E	E	E	E
addr	1	pad	pad	1	8	15	pad	pad	2	9	16	pad	pad	3	10	17	E
addr	2	pad	pad	3	10	17	pad	pad	4	11	18	pad	pad	5	12	19	E
addr	3	pad	pad	5	12	19	pad	pad	6	13	20	pad	pad	7	14	21	E
addr	4	pad	pad	7	14	21	pad	pad	pad	pad	pad	pad	pad	pad	pad	pad	E

Fig. 9. Max pooling data layout with window size of 5x5.

the previous round. Considering the following situation with input activations of size 5x5 and RAFFT with 16 input ports. This means that there are in total 25 inputs need to be fed into the RAFFT, so we have to complete this computation in two rounds. As per our design choice, we first feed in 9 input activations and 16 in the second round. After the first round, we retain the result of max pooling among the 9 input data inside the RAFFT and we call that residue and then stream in the next 16 data and go through the same steps to finish the comparison of all 25 input activations at the last stage. In the next subsection, we will cover a detailed walk-through example of max pooling within the RAFFT architecture.

2) Walk-through Example of Max Pooling: In this subsection, we give a walk-through example of using RAFFT to perform max pooling when both input activations and output activations share the same data layout. In this walk-through example, we have input window size of 5x5 and the RAFFT can take in 15 inputs at a time, so the data mapping will be like Figure 9.

In Figure 9, it shows the mapping of a single max pooling window on the RAFFT. The overall dataflow is the input data traversing through all the activated nodes inside the RAFFT till the last stage. In the last stage, it will have to register the partially generated results, or residue. At the beginning, in Cycle 0, the reduction phase is achieved by activating Max-0 and the last switch retains the reduced input with the Through switch, referring to Figure 8. In the next cycle, the reduction phase is maintained as is, and now the RAFFT will operate on the previously held data with the newly/presently generated data with the Max-through and the residue activated, which is shown in Figure 12. Thus, the RAFFT not only retains residue, but also uses them in the next phase of its operation.

This feature results in considerable data reuse. Take for example the shifted max pooling window which needs to operate on another batch 25 elements from Row 1-2, but in fact shares 15 elements from the previous data. So, in this scenario, the RAFFT would store the residue in the corresponding internal register that is connected to the expected output port. The RAFFT operates by moving the max pooling window from left to right, till the output is reached, after which the RAFFT repeats the same sequence of operations on the remaining inputs of the current layer.

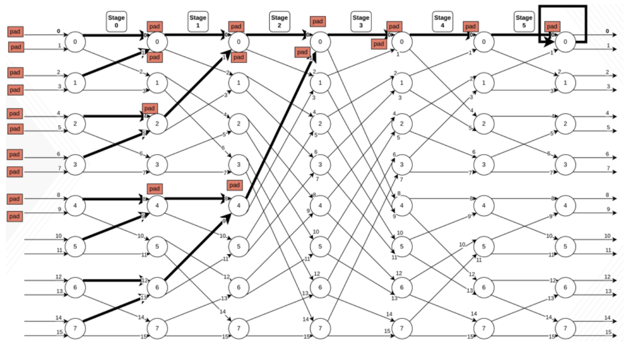


Fig. 10. Max pooling data layout with window size of 5x5.

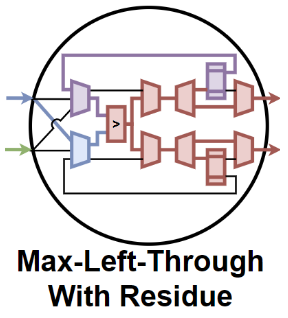
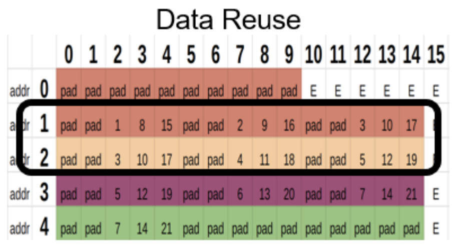


Fig. 11. Max-through Switch with the Residue on the left activated.



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
addr	0	pad	pad	pad	pad	pad	pad	pad	pad	pad	pad	E	E	E	E	E	E
addr	1	pad	pad	1	8	15	pad	pad	2	9	16	pad	pad	3	10	17	[illegible]
addr	2	pad	pad	3	10	17	pad	pad	4	11	18	pad	pad	5	12	19	[illegible]
addr	3	pad	pad	5	12	19	pad	pad	6	13	20	pad	pad	7	14	21	E
addr	4	pad	pad	7	14	21	pad	pad	pad	pad	pad	pad	pad	pad	pad	pad	E

Fig. 12. Max-through Switch with the Residue on the left activated.

BN Implementation	Latency	DSP Usage	FF Usage	LUT Usage
DSP	2.66e + 03	128 (7%)	2900 (∼0%)	14483 (6%)
1-level Multiplier w/o unroll	6.983e + 04	64 (3%)	8440 (1%)	32528 (14%)
1-level Multiplier + unroll	2.68e + 03	64 (3%)	41352 (8%)	83393 (36%)
2-level Multiplier + unroll	2.66e + 03	64 (3%)	12356 (2%)	76417 (33%)

TABLE I
PERFORMANCE AND RESOURCE UTILIZATION FOR BATCH NORMALIZATION

V. EVALUATION

To evaluate our work, we implement the submitted C++ testbench, named maeri_v2_1_tb.cpp, to validate the functional correctness of Batch Normalization, ReLU, and Max Pooling. In addition, we set up the performance and resource utilization tests on our Arbitrary IFC-Buffer Burst Transfer and Batch Normalization. We used Xilinx ZCU104 board for this evaluation. To measure the performance boost obtained from the burst transfer, we set the size of data transfer through the AXI ports to the on-board buffers to be 2000 words. The runtime result shows that the origin baseline has a latency of 280.76 microseconds in transferring 2000 words of data, while our bust mode transfer can cut the execution time to 89.43 microseconds. In general, the burst mode transfer gives us a 3.5x speedup as compared to our baseline of non-burst mode transfer.

For evaluating our Batch Normalization implementations, we compared the resource utilizations and latency differences from the computations done by DSPs and customized multipliers. Table I summarizes the results of different Batch Normalization implementations. We observe that the best-case latency is around 2.66e+03 which can be engendered by using either DSP and 2-level parallelized multiplier with unrolling enabled. Another key contribution of our optimizations is that it provides flexibility in resource utilization. The MAERI 2.0 design keeps computations of convolution, Batch Normalization, ReLU, and Max Pooling all on the PL side, so with the pipelined operation, it is crucial to consider the resource utilization of different layers carefully. With a flexible implementation of Batch Normalization, we are able to use the multi-multiplier when the DSPs are the limiting resources and choose DSP as the primary implementation if FFs and LUTs are scarce.

VI. CONCLUSION AND FUTURE WORK

In this work, we used HLS to accelerate MAERI 2.0 by supporting various layers. To leverage the flexible architecture and data layout of the MAERI 2.0, we make the extensions including Arbitrary IFC-Buffer Burst Transfer, ReLU, Batch Normalization, and Max Pooling with the RAFFT network. The purpose is to try to keep as many computations as possible on the PL side so that it can minimize the PS-PL communication overhead between layers. The result shows that our burst transfer provides a 3.5× speedup compared to the non-burst transfer. Our customized multiplier can offer more flexibility in resource utilization without loss of performance.

For the future work, we can expand the applications of our RAFFT network to the other computations. For example, we can use the data layout similar to max pooling to implement average pooling. With the given input activations, average pooling follows the similar logic to max pooling, but we need to extend the switching nodes to do division and addition more than a simple comparator.

References

[1] "maeri-project," https://maeri-project.github.io/.

[2] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[3] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, "Dissecting the graphcore ipu architecture via microbenchmarking," 2019. [Online]. Available: https://arxiv.org/abs/1912.03413

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[5] S.-C. Kao and T. Krishna, "Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[6] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[7] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *2017 Eleventh IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2017, pp. 1–8.

[8] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 461–475. [Online]. Available: https://doi.org/10.1145/3173162.3173176

[9] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 304–315.

[10] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 58–68.