Accelerator Templates and Runtime Support for Variable Precision CNN

Srivatsan Krishnan[1], Piotr Ratusziak[1], Chris Johnson[1], Duncan Moss[2], and Suchit Subhaschandra[1]

[1]Intel Corporation
[2]University of Sydney

***Abstract*— Deep learning algorithms are playing an important role in computer vision and image recognition tasks. At the very outset, one can view neural networks as an function approximator. Recent trends in deep learning involves exploring lower precision operations for increasing performance and energy efficiency. To this end, In this paper we provide variable precision accelerator template in FPGA and its corresponding software and runtime stacks that interfaces with these accelerators. We first implement the prototype CNN template to support FP-32 based weights and activations. We then extend the design to support Int16, Int8 ,Int4 ,1-bit and ternary weights with Fp-32 and Int8 activations in Intel HARPv2 platform. These templates can be extended to any arbitrary precisions. We then design runtime and API for dynamically reconfiguring the precision. In terms of performance and energy efficiency, these variable precision template accelerators provides (24 Tops, 545.5 Gops/W),(6.3 Tops, 141 Gops/W), (3.1 Tops, 70 Gops/W) and (1.6 TOPS, 40 Gops/W) for binary, Int4, Int8 and Int16 precisions respectively. To the best of our knowledge, these measured numbers are the state of the art for Arria 10 class of FPGA devices.**

I. Introduction

CNN's are making huge strides into solving challenging problems in computer vision and image recognition task with near human level accuracy. Alexnet [1] achieved an accuracy of 84.6% compared to state of the art ImageNet classifier in 2012 time frame. One of the major thrust in achieving that level accuracy was availability of high performance and parallel computer architectures. Since then, the deep learning networks have become much deeper causing the compute requirement to explode. Alexnet had 5 layers of convolution. The next state of the art ImageNet classifier VGG [2] had 19 layers. The current state of art ImageNet classifier, ResNet-151 [3] achieved an accuracy of 96.3% and the network topology was 151 layers deep. It is evident that as the network gets deeper, the computational requirement also increases significantly. We can see this trend in Fig.1.

Another interesting trend in deep learning is that weights and activation can be represented in lower precision and still achieve near human level accuracy. This area of research is still very active and we have seen the effect of lower precision arithmetic, proliferate into various compute architectures. Many architectures like Nvidia GPU's [4] [5] and Google TPU's [6] [7] have native support for lower precisions. The compute density increases as precision decreases. Fig. 1 also shows this trend. To take it further, there has been prior work that characterized accuracy and performance beyond 8-bit [8] [9]. For example, XNOR-Net [10] used 1-bit to represent weights and activations and still achieved the same level of classification accuracy as the classic AlexNet topology. But these prior work computed various layers at same precision. Another interesting work by Judd et al [11] proposed having different precision per layer and an architecture to perform bit serial compute , where they showed it can increase performance linearly by using just the right amount of precision.

Couple of interesting and pertinent question arises due to this trend:

- *Are we computing more than what is required?*
- *Does each layer in a neural network topology has a definitive percision that achieves near human level accuracy?*
- *Can computer architects exploit this trend(definitve precision per layer) for achieving better performance and energy efficiency?*
- *Is there an architecture that can allow programmers to fix precision based on need rather than one size datapath that fits all?*
- *What kind of architecture support and runtimes are needed for programming languages and machine learning frameworks?*

At the very outset, CNN's can be viewed as function approximators and are very resilient to precision errors. Our contribution is to provide infrastructure components, that includes a variety of variable precision accelerator templates and corresponding runtime software, that allows machine learning researchers and data scientist to answer some of the above questions. For instance, instead of running the entire network on a 8-bit precision, ML researchers can now use our dynamic configuration API's to change the precision at per layer basis and see how it affects the overall accuracy.

II. Hardware Accelerator Templates

Hardware accelerator for CNN is modular and based on a template design. Briefly, there are three parts to the design as shown in Fig.2. (1) **Configurable Memory Bank:** First part of the template is the on-chip memory interface logic that talks to the data management unit. It is basically a

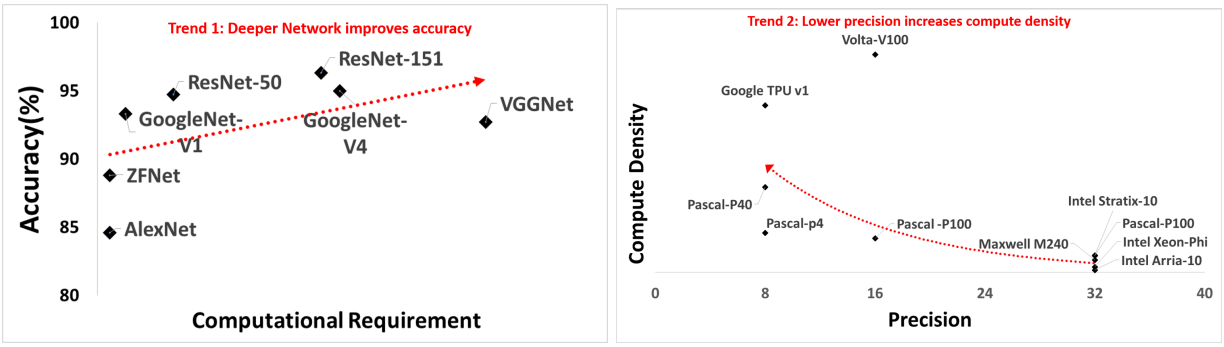

Fig. 1: Major Trends in Deep learning impacting Computer architecture. Firstly as deeper networks achieves higher accuracy. But as the network becomes deeper, the computation requirement also increases. Secondly, CNN's are resilient to precision errors and lower precision network are gaining prominence for their lower model size. In terms of compute, lower precision compute provides higher compute density and energy efficiencies.

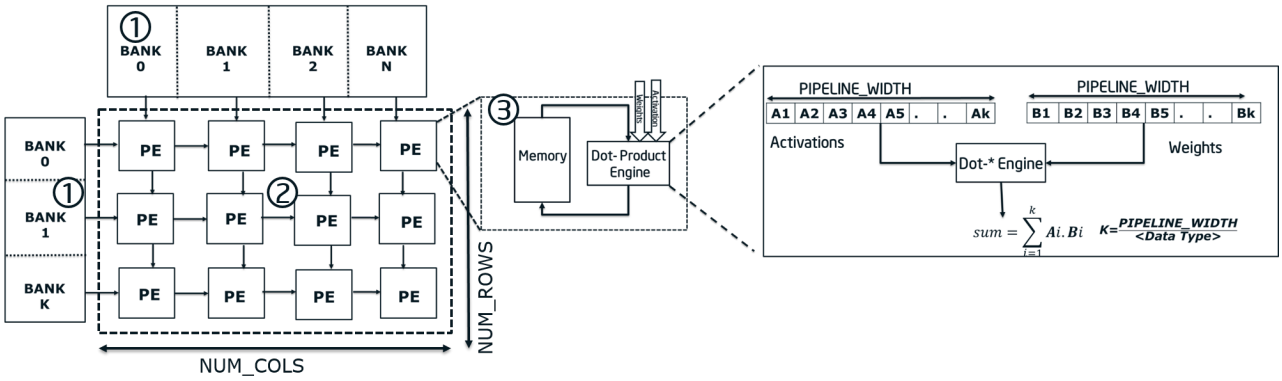

Fig. 2: Hardware accelerator template for variable precision CNN. The First part of the template is the multi bank memory for storing activations and weights. The second part of the design is 2D systolic array of PE. Each PE has an accumulator and dot product engine. The pipeline register that feeds to the dot product engine is of definitive width PIPELINE_WIDTH. For example the PIPELINE_WIDTH can be 256bits, which means we can pack 8 elements of FP32, 16 elements of INT16 and 32 elements of INT8 in the pipeline registers. The Dot-* engine will change depending upon the precision. The Dot-* engine will use both DSP as well as ALM's available on the FPGA to maximize the compute density

controller that issues requests to the load/store interface to fetch data from the system memory. It also comprises of on-chip memory buffers to store the data locally on the FPGA. The memory banks are configurable to allow different blocking schemes to maximize data reuse. Depending upon the precision, the packing in the on-chip memory changes. Packing schemes, memory blocking and degree of data reuse are all configurable at runtime and are exposed to the runtime software. Runtime software utilizes the configurable registers to set the configurations. Configuration registers are basically memory mapped register files, which are used by the controller to configure the mode and functioning of the accelerator. ② **2-D Systolic Array:** The second important component is the 2-D systolic array that consist of processing elements(PE). Each PE can be uniquely identified by a Row and Column ID. The number of PE's in the accelerator can be increased or decreased by varying the number of rows and columns in the 2-D systolic array. ③ **Processing Engine(PE):** Each PE is configured to perform Multiply and Accumulate(MAC) operations on two vectors(weights and activations are decomposed into vectors). The pipeline registers that feeds to the MAC units are of configurable width. For this implementation, we fixed the "PIPELINE_WIDTH" parameter to 256 bits. This gives us the ability to pack 8, 16, 32, 64 and 256 elements of float, Int16, Int8, Int4 and 1-bit numbers respectively. Alternatively, one can view pipeline register as AVX registers that feeds to the AVX units in Intel CPU's. But unlike AVX registers, where the minimum granularity is 8-bit, our accelerator template can be of any arbitrary width. The template design maximizes the use of both DSP's and ALM's available in the FPGA. The compute density can be maximized by adding additional row of PE's to the 2-D systolic array grid. It is to be noted that number of rows and number of columns used in the 2-D systolic array grid is same for all the precision modes. We fixed the "NUM_ROWS" and "NUM_COLS" of the 2-D systolic array to be 10 and 16 for evaluating different precision modes. For lower precision implementations(Int8,Int4,etc), we still have lots ALM's available to increase the compute density.

III. HARPv2 Platform and Template Integration

The accelerator template and runtime system is implemented in Intel HARPv2 platform. Intel HARPv2 [12] is a first of a kind platform that tightly couples the FPGA and a high performance Xeon processor in a single package. The FPGA used in the multichip package is Intel Arria 10. One of the key features of the platform is that, the accelerator built on the FPGA has coherent access to Xeon's Last Level Cache(LLC) and system memory. A High level platform architecture of HARPv2 is shown in Fig.3. The Intel BBS corresponds to the interface logic that abstracts the physical QPI and PCIe links to the Xeon. It also provides a simple load-store like interface called CCI-P to the accelerator to access system memory. Integration of hardware accelerator template to the Intel Harpv2 platform is shown in Fig.3.

(1) **Interfacing hardware accelerator template with DMU Unit:** One of the key component to the hardware template is the memory banks. Each of the memory bank has 2 ports: read port and write port. The memory bank write interface is 64Bytes(Same as Intel Xeon's cacheline width). DMU's RX port is connected to the write port of the template memory bank. The read port of the memory bank is 32Bytes. 2-D systolic array has a small centralized controller(not shown in Fig. 3.This controller issues read signal to the template memory banks. This controller also orchestrate when 32Bytes of data is read from the memory banks. It is to be noted that the data is fed only to the PE's that are in the first row and first column. Each PE processes the input data and also propagates the same data and control signals to the neighboring PE's.

(2) **Interfacing DMU to CCI-P:** CCI-P has three main ports: TX, RX and MMIO. TX port provides control and data signals to write to Xeon's system memory from the accelerator. The RX port provides control and data signals to read from Xeon's system memory. The data signal width in TX port is of cacheline granularity. DMU unit uses the RX port to read the data from system memory and store it in the hardware accelerator template's memory bank. It uses the TX port to write the result from the 2-D systolic array to Xeon's system memory.

(3) **Interfacing CCI-P to Configuration Registers:** Runtime software uses MMIO ports to set the configuration registers. Configuration registers are memory mapped IO, which means these registers are visible in the system address space. The CCI-P provides MMIO ports to both read and write data into these registers.

(4) **Interfacing Performance counters to CCI-P**: Performance counters are used for storing runtime performance metrics of the CNN accelerator. Performance counters include, number of clock cycles from start to end of a run, number of idle cycles, number of cycle the systolic array stalled because of the read bandwidth , number of clock cycle the systolic array stalled because of write bandwidth etc. These counters are used to profile the accelerator during runtime. The values in the performance counters are written to the configuration registers. The runtime software uses MMIO port of CCI-P to read back the values stored in these configuration registers.The runtime software can then use these performance counters to calculate metrics like FLOP/Sec ,TOP/Sec, read bandwith and write bandwidth etc.

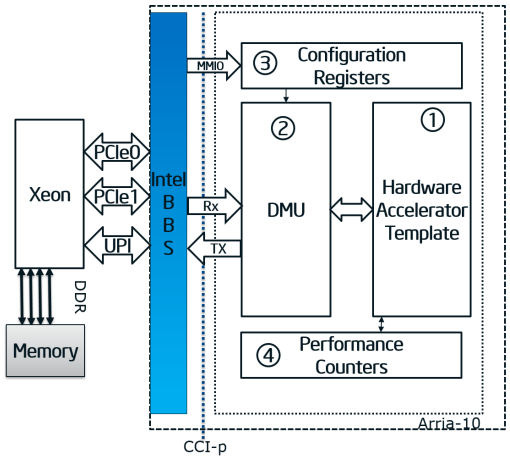

Fig. 3: Platform architecture of Intel HARPv2

IV. Runtime Support

The hardware accelerator template has various configurable parameters that are exposed to runtime software. The CNN application uses the Intel HARPv2 user mode runtime and kernel driver to set these configurations. Intel HARPv2 comes with its own driver stack called Intel Accelerator Abstraction Layer(Intel AAL). The integration of runtime software with the hardware accelerator template is shown in Fig.4. There are 4 key components in the integration.

(1) **Application API:** The main compute pattern in convolution and FC layers in a deep learning topology is GEMM(General Matrix Multiplication). The Application exposes GEMM API for various precision. The API's are templated to support different precisions and modes. Depending upon the precision, the number of elements packed into cacheline also changes. The API is shown in Listing 1. "a_rows" and "b_cols" refers to the number of rows and columns in A and B matrix respectively. "common" parameter refers to the common dimension in both the matrix. "i_alpha" and "i_beta" refers to the scaling parameters and "i_mode" refers to the mode in which the hardware accelerator template is set. Internally, the application API uses AAL user mode runtime to access and initialize the FPGA device.

Listing 1: API to access hardware template

```
template<typename T1, typename T2>
fpga_gemm<T1,T2>::fpga_gemm( uint32_t a_rows,
                             unit32_t b_cols,
                             unit32_t common,
                             float i_alpha,
                             float i_beta,
                             GEMM_MODE i_mode)
```

There is also a dynamic configuration API available to change the template from one precision to another. The templates are basically a precompiled bitstreams and the AAL service internally uses partial reconfiguration to switch from one mode to another. The API for dynamic configuration is shown in Listing 2. In future, we plan to have a hybrid 2-D systolic array where we can have mixed precision PE between different rows or columns and exposing those configuration to runtime software. That will eliminate the time it takes to do partial reconfiguration of FPGA device.

Listing 2: API to dynamically configuring templates

```
int config_afu_sgemm(const char *pathname)
{

gemmAAL<int, int> hardware_template;

hardware_template.setHW(true);
return hardware_template.configSGEMM(pathname);
}
```

(2) & (3) **Intel Accelerator Abstraction Layer(Intel AAL):** AAL layer provides the necessary runtime services and API to access the FPGA device. The CNN application API is built on top of AAL user-mode API's and services. At a very high level, AAL services can be briefly classified into two categories:

- ***AAL user-mode runtime:*** *These are interfaces that abstract FPGA hardware via a service oriented model. Various services in AAL user-mode runtime can be aggregated to build application specific services.*
- ***AAL kernel-mode driver:*** *These includes interfaces for allocation of Direct Memory Access(DMA) buffers with shared addressing between hardware accelerator template and user application. It provides interfaces to access Memory Mapped IO(MMIO) registers in the hardware template. It also provides interfaces to perform partial reconfiguration on the FPGA device. The dynamic reconfiguration API uses AAL kernel-mode driver's interface under the hood.*

AAL kernel mode driver utilizes Intel BBS to enumerate the device.

(4) **Intel BBS :** Intel BBS is the infrastructure shell component in the FPGA. It abstracts UPI(Intel's Coherent link to Xeon) and PCIe links and provides a simple load-store like interface called CCI-P to the user accelerator. Intel BBS also has AAL kernel visible MMIO registers. The AAL kernel driver uses these configuration registers for FPGA device enumeration and initialization. Apart from that, Intel BBS also has controllers for doing partial reconfiguration of the FPGA device.

V. RESULT AND DISCUSSION

We first implement a baseline hardware accelerator template for CNN in FP-32 data type. We then extend the same design to support Int16, Int8, Int4, 1-bit. We also extend the design to support ternary operation where the activation is

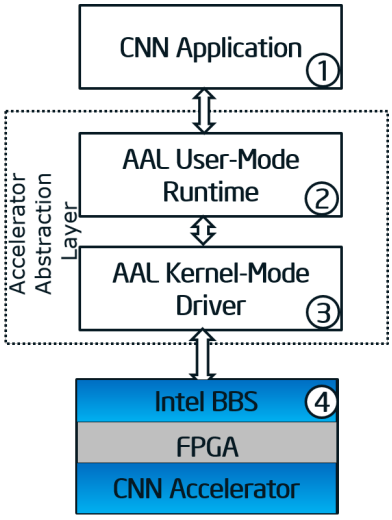

Fig. 4: Runtime software stack for hardware accelerator template for CNN.

in FP-32 or Int8 and the weights are 2-bits(-1,0,+1). Fig.5

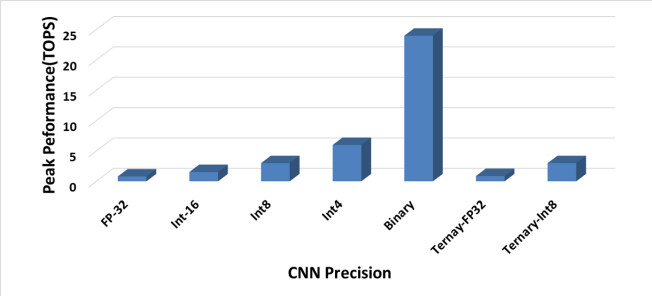

Fig. 5: Performance comparison of various modes supported in the hardware accelerator template.

shows the performance comparison of various precision modes in CNN accelerator template. The trend is consistent that, as we move towards lower precision, we observe linear scaling in performance. In terms of raw performance, we see 32x improvement for binary, 8x improvement for Int4, 4x improvement for Int8 and 2x improvement for Int16 compared to floating point version. We don't see any performance gains in ternary versions compared to their FP-32 and Int8 counterparts. But we do save on-chip memory and FPGA resources. In future, we plan to optimize the ternary design to pack more PE's to improve compute density. We can also plan to optimize the memory bandwidth and reduce the load times for weights.

Fig.6 shows the performance per watt of various precision modes in CNN hardware accelerator template. In terms of energy efficiency, We achieve 1.6x improvement for Int16, 3.2x improvement for Int8, 6.24x improvement for Int4 and 23.82x improvement for binary compared to floating point implementation. All these measurement was made at the convolution API's from the AAL level.

VI. CONCLUSION

In this paper, we proposed a variable precision accelerator architecture template and its supporting runtime for machine learning researchers. This infrastructure can be used by the machine learning researchers to determine till what point we can approximate the precision without losing performance. In terms of performance, we measured 24Tops for binary, 6.3 Tops for Int4, 3.1 Tops for Int8, 1.6Tops for Int16 which translates to 32x, 8x and 4x and 2x improvement in performance compared to floating point implementation. The ternary version with 2-bit weights and 32 bit activation achieved 0.8TFLOPS and 2-bit weights and 16-bit activation achieved 3TOPs for peak performance. In terms of energy efficiency, we measured 545GOPS/W, 141 Gops/W, 70 Gops/W, 40 Gops/W for Binary, Int4, Int8 and Int16 respectively. The energy efficiency obtained also shows 2x improvement for Int16, 4x improvement for Int8 , 7x improvement for Int4 and 23.82x improvement compared to floating point implementations. The ternary variant with 8-bit activations and 2-bit weights shows an improvement of 3.6x compared to floating point implementation.

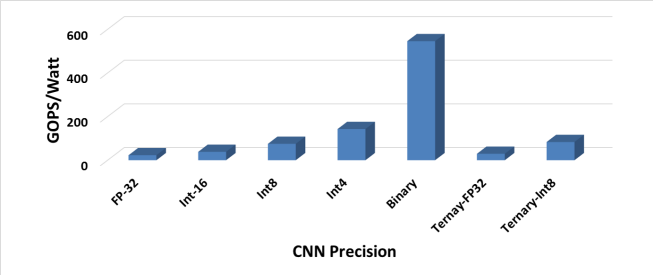

Fig. 6: Performance comparison of various modes supported in the hardware accelerator template.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," pp. 1097–1105, 2012. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[4] "Nvidia tesla p4 inferencing accelerator," http://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf, accessed: 2017-08-30.

[5] "Nvidia tesla p4 inferencing accelerator," https://www.nvidia.com/en-us/data-center/tesla-v100/, accessed: 2017-08-30.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc 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, 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, and J. Ross, "In-datacenter performance analysis of a tensor processing unit," 2017. [Online]. Available: https://arxiv.org/pdf/1704.04760.pdf

[7] "Nvidia tesla p4 inferencing accelerator," https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, accessed: 2017-08-30.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[9] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1604.03168, 2016. [Online]. Available: http://arxiv.org/abs/1604.03168

[10] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016. [Online]. Available: http://arxiv.org/abs/1603.05279

[11] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[12] "Intel hardware accelerator research program," https://www.fpl2017.org/conference/activities/14/intel-hardware-accelerator-research-program-a-tutorial-for-learning-and-using-the-intel-xeon-with-in-package-fpga/, accessed: 2017-08-30.