Application of Convolutional Neural Networks on Intel® Xeon® Processor with Integrated FPGA

Philip Colangelo[xy], Enno Luebbers[x], Randy Huang[x], Martin Margala[y] and Kevin Nealis[x]

[x]Intel Corporation
San Jose, USA
philip.colangelo|enno.luebbers|kevin.nealis
@intel.com

[y]University of Massachusetts Lowell
Lowell, USA
Martin_Margala@uml.edu

***Abstract*— Intel®'s Xeon® processor with integrated FPGA is a new research platform that provides all the capabilities of a Broadwell Xeon Processor with the added functionality of an Arria 10 FPGA in the same package. In this paper, we present an implementation on this platform to showcase the abilities and effectiveness of utilizing both hardware architectures to accelerate a convolutional based neural network (CNN). We choose a network topology that uses binary weights and low precision activation data to take advantage of the available customizable fabric provided by the FPGA. Further, compared to standard multiply accumulate CNN's, binary weighted networks (BWN) reduce the amount of computation by eliminating the need for multiplication resulting in little to no classification accuracy degradation. Coupling Intel's Open Programmable Acceleration Engine (OPAE) with Caffe provides a robust framework that was used as the foundation for our application. Due to the convolution primitives taking the most computation in our network, we offload the feature and weight data to a customized binary convolution accelerator loaded in the FPGA. Employing the low latency Quick Path Interconnect (QPI) that bridges the Broadwell Xeon processor and Arria 10 FPGA, we can carry out fine-grained offloads while avoiding bandwidth bottlenecks. An initial proof of concept design showcasing this new platform that utilizes only a portion of the FPGA core logic exemplifies that by using both the Xeon processor and FPGA together we can improve the throughput by 2x on some layers and by 1.3x overall.**

Keywords—Xeon, FPGA, BWN; CNN

I. INTRODUCTION

Intel Xeon Processor with Integrated FPGA, denoted as multichip package (MCP) in this paper, is a new research platform by Intel that provides all the capabilities of a Broadwell Xeon Processor with the added functionality of an Arria 10 FPGA in the same package. This marriage proves to be a powerhouse as the Xeon processor provides computing power for a variety of software workloads and Arria 10 with its reconfigurable fabric serves as a vehicle for targeted, specific workloads.

Common application development for FPGA includes a system level architecture with a discrete card connected through PCIe motherboard slots. In this arrangement, workloads are typically offloaded in coarse-grained fashion to overcome complexities in maintaining memories between host and accelerator and to accommodate PCIe bandwidth constraint. The MCP breaks this constraint by providing a system capable of meeting fine-grained, low latency, and coherent workload demands.

Application development for MCP is relatively new and few people have the knowledge in how to do it. Enabling others to develop MCP applications will provide a highly flexible platform capable of accelerating a variety of workloads including machine learning, compression, network processing, video processing, and many more. In this paper, we document the journey we took in creating an application for MCP and provide the following contributions:

- Design process that explains the steps taken to develop the necessary software framework, communication system for data flow, and customized hardware used for accelerating a specific workload on MCP.
- Example design of accelerating a BWN along with initial results.

Our contribution of a BWN accelerator displays the capability of the MCP platform by speeding up in a fine-grained fashion the layers that take the maximum amount of time. We prove that the MCP platform is an ideal fit for fine-grained acceleration designs in that the integrated FPGA shows up to a 2x improvement in some layers over the stand-alone CPU. This style of acceleration for FPGA was not easily accessible until now.

The rest of this paper is organized as follows: Section 2 provides an overview of application development on MCP hardware, software, and overview of BWN. Section 3 details the application case study of a BWN accelerator targeting the MCP. Finally, section 4 delivers the evaluation and results of our design.

II. APPLICATION DEVELOPMENT

A. Intel Xeon Processor with Integrated FPGA

Shown in Fig. 1. is the MCP which contains a Broadwell Xeon (BDX) processor [1] with an integrated Arria 10 GX 1150 FPGA [2]. The BDX is a server grade processor typically found in the data center that provides first class performance across a wide range of software enabled, general purpose workloads. BDX workloads can include supporting cloud computing, high

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

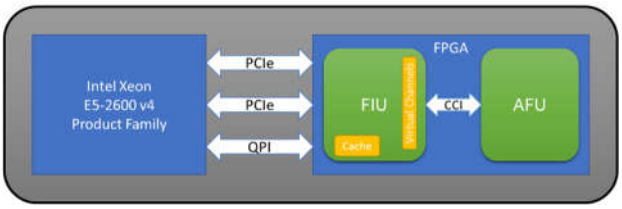

Fig. 1. Intel Xeon Processor with Integrated FPGA

performance computing, network processing, and storage solutions. In some cases, workloads require acceleration beyond what a general-purpose processor can offer. The MCPs integrated FPGA is designed to bridge this gap by providing hardware acceleration for specific workloads through its reconfigurable logic. Packed with 1,518 DSP blocks supporting the industry's first hardened single precision floating point units [3], 1,150,000 logic elements, and 54,260 M20K blocks, the Arria 10 GX 1150 is an ideal choice for many high-performance applications.

Communication between the Xeon processor and FPGA can be established through Intel's Quick Path Interconnect (QPI) [4] and/or two PCIe Gen3x8 interfaces. These channels provide flexibility for the developer when determining what interface is required for their workload. Developers have the option to choose between QPI or PCIe to take advantage of either protocol's specific features, or combine the two with automatic arbitration to obtain a theoretical peak bandwidth up to 28.8 GB/s. QPI is a high speed, low latency, coherent interconnect developed by Intel for links in and out of the processor. Targeted for low latency workloads, QPI offers up to 12.8 GB/s bandwidth. Smaller sized workloads transfer most efficiently through QPI due to lower overhead compared to PCIe. Further, cache coherency provides the ability for acceleration of applications that require memory between the processor and accelerator to be maintained. Two PCIe Gen3x8 interfaces provide a combined maximum bandwidth of 16 GB/s. Many discrete FPGA designs, especially for data center applications, use this configuration making discrete to MCP porting natural. Having access to both QPI and PCIe provides a solution platform for a myriad of design problems.

The FPGA side of the MCP is broken up into two regions: the FPGA Interface Manager (FIM) and Accelerator Functional Unit (AFU). The FIM is a logic region provided by Intel that implements QPI and PCIe communication between BDX and the AFU. The AFU is the user defined accelerator code. These two blocks communicate through a Accelerator Function Interface (AFI) managed by the FIM that provides a unified memory model interface to the AFU. AFI is a 64-byte link abstracted as a series of 4 virtual channels allowing designers to choose between both PCIe interfaces, QPI, or all 3.

AFUs can be designed in either high (OpenCL) or low (HDL) levels of abstraction and follow a similar design methodology compared to discrete platforms. The major difference between discrete designs and MCP designs are found in off-chip memory accesses. Discrete designs commonly see large workloads transferred over PCIe that are too large to fit entirely on-chip and rely on attached DDR for temporary storage. Sharing DDR with the Xeon processor places the AFU in closer partnership that opens the opportunity to accelerate problems in a different way. For example, leveraging the QPI interface allows for fine-grained offload acceleration that will come in handy during our case study.

Fig. 2 provides the HDL flow for designing applications targeting the MCP. Host applications are written in C/C++ for the Xeon and compiled down to an executable. This executable will run the user's application and provide control flow for the FPGA through OPAE. More detail on OPAE will be provided in section II.B. On the accelerator side, Quartus is used as the design tool for developing Intel FPGA projects. In this flow, Quartus is used for design entry, synthesis, place and route (PAR), and compiling to the AFU bitstream. For more information on these steps please refer to [5].

The second method for developing AFUs is to use a high-level design language. Intel FPGA SDK for OpenCL [6] is a way to enable software developers to create accelerator applications in both discrete and integrated platforms. Fig. 3 depicts the high-level flow for OpenCL development for MCP. Like standard OpenCL projects, users will create OpenCL kernels that define the inputs and outputs along with the algorithm software model for their specific needs. Kernels are then compiled using the Intel FPGA for OpenCL Offline Compiler (AOC) which generates the AFU bitstream. AFU bitstreams are loaded into the FPGA through the OpenCL host code API.

Running applications on the MCP begin by loading the user defined bitstream onto the FPGA. After power on, AFU designs can be partial reconfigured (PR) on the FPGA using the OPAE SDK API either as a standalone executable or during application runtime. PR allows for specific regions of the FPGA to be reconfigured while maintaining state for all other regions. For deployment, FPGA images that include both FIM and AFU bitstreams can be stored in flash memory and read during CPU power on.

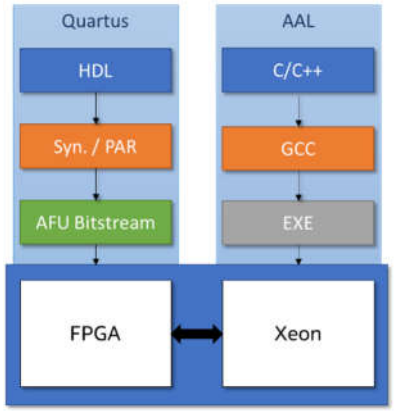

Fig. 2. HDL development stack for MCP

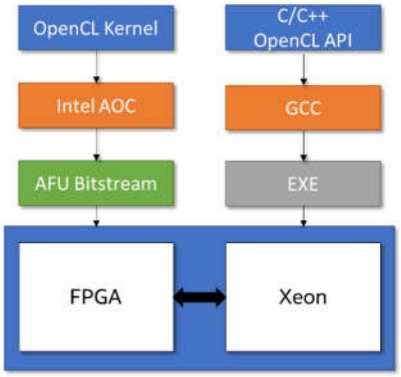

Fig. 3. OpenCL development stack for MCP

B. Open Programmable Acceleration Engine

1) Overview

The Open Programmable Acceleration Engine (OPAE) abstracts the hardware/software interface of the programmable accelerator resources and application software. It is designed as a software stack that can support a multitude of accelerator technologies. The OPAE Software Development Kit (OPAESDK) provides a service-oriented, C++-based development framework which presents *services* to application software components; the implementation of these services is decoupled from their interface, allowing transparent support of accelerated hardware resources. Among the infrastructure for the definition and implementation of services, OPAE also provides mechanisms for run-time reconfiguration of the service composition, e.g. depending on resource availability or application requirements.

As such, OPAE is not a software technology that's aimed specifically at FPGAs, but it provides distinct features which fit a platform with dynamically programmable hardware:

- The OPAE driver stack allows lightweight creation and destruction of accelerator resources, which caters to the dynamic nature of partially reconfigurable FPGAs
- OPAEs service-oriented decoupling of interfaces (capabilities) and implementations allow transparent run-time selection of accelerated or non-accelerated implementations, based on resource availability (even across different accelerator technologies)
- OPAE features a resource management interface which promotes user-defined policies for resource control and assignment

Many of these features are targeted at large-scale deployments of accelerated workloads in heterogeneous environments. However, OPAEs architecture also scales down to single nodes with single applications, such that the same development model can be leveraged in research and development as well as multi-node deployment.

2) Service-Oriented Development Framework

OPAEs service-oriented software framework is supported by a run-time library (the "OPAE Runtime"), which is linked into an application using it, and provides the mechanisms to discover and match services and implementations, as well as run-time loading of service modules. In OPAE, service components are linked through interfaces; they define a service's capabilities and are used to call the service-specific APIs. Interfaces are pure abstract C++ classes which are bound to service implementations at run-time. Since services can in turn make use of other services' capabilities themselves, this framework allows flexible and transparent composition of services to provide a desired functionality.

Consider the simple example outlined in Fig. 4. Here, a set of capabilities of an abstract service "Foo" is published through interface "IFoo", which a client application "Bar" accesses. Bar does not need to know the implementation of Foo at compile-time; it can be bound to either an FPGA-accelerated version (FooFPGA) or a software implementation (FooSW) by OPAEs resource management at run-time, when Bar allocates the "Foo" service. If IFoo features an asynchronous access protocol, Bar also needs to implement a client interface "IFooClient" (which will usually define callback functions that are called when an operation initiated through IFoo completes).

OPAE also includes several generic service interfaces for service construction and release, which are omitted from the figure for simplicity. In fact, the Runtime is also a service, to which the calling application is a RuntimeClient.

3) FPGA Accelerator Link Interface

At the lowest level, services that want to provide an FPGA accelerated implementation will need to get access to FPGA resources like AFUs through means like control registers, shared memory, and other communication and control channels. To facilitate this, OPAE provides an implementation of the AFU Link Interface (ALI), basically forming the counterpart to the physical AFI interface within the FPGA. ALI provides functions to map an AFUs control register space into application memory, allocate shared DMA buffers, configure and reset FPGA resources, among others. It publishes these features via a collection of interfaces shown in TABLE I. Services (or applications directly) can allocate an ALI service, request access to a resource, and use the provided API to directly interact with the resource's features; the assignment and management of the resource as well as the

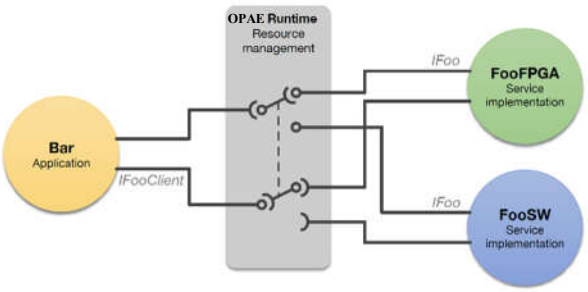

Fig. 4. OPAE service-oriented development example

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

routing of communication and control between the service and the FPGA is transparently handled by the OPAE infrastructure and drivers.

TABLE I. *EXAMPLE OPAE API*

Interface	Functionality
IALIMMIO	Access AFU registers
IALIUMSG	Low-latency messaging
IALIBuffer	Allocate shared memory
IALIPerf	Performance counters
IALIReset	Reset AFU
IALIReconfigure	Reconfigure AFU
IALIReconfigureClient	Callbacks for IALIReconfigure
IALISignalTap	AFU debugging
IALIError	Error reporting
IALITemperature	Temperature monitoring
IALIPower	Power monitoring

C. Debugging MCP Applications

Part of the Intel FPGA SDK for OpenCL is the Intel FPGA SDK for OpenCL Emulator that provides support for functional verification. During emulation, the compiler compiles your kernel but instead of a full PAR, the compiler generates an aocx file that contains a software version of the design. A typical compilation time for generating aocx files is in the order of minutes while a full PAR compile can take multiple hours. Emulation provides a platform for verification without ever running on actual hardware. This gives designers feedback within minutes useful for the next design iteration. The Intel FPGA SDK for OpenCL provides support for debugging kernels that is like GDB debugging because it allows for code stepping, breakpoints, and printing variables.

By taking advantage of the run-time binding of interfaces to implementations, OPAE provides a convenient and effective AFU Simulation Environment (ASE) to aid accelerator development. ASE is a client-server development and debug environment used to simulate MCP applications. This tool is provided as part of the OPAESDK. If directed appropriately, ALI will not initiate a driver connection to allocate FPGA resources for an AFU, but instead invoke a user-mode IPC link to an RTL logic simulator and relay all ALI transactions to that simulation model, effectively allowing quick verification of both software and accelerator RTL together. Applications verified using ASE are expected to port seamlessly to the MCP hardware. In similar methodology to the OpenCL emulator, ASE provides a system for fast design iterations providing cycle accurate hardware verification through simulation without the need for full hardware compiles. If verification passes but a fully compiled design fails, e.g., it stalls or provides incorrect results, then bitstreams can be debugged using SignalTap. Provided as part of Quartus Prime software, SignalTap is a debugging tool that allows signals to be displayed in real time. A specific flow is provided as part of MCP development that instructs the designer in how to enable SignalTap to observe specific signals and timing diagrams with SignalTap Logic Analyzer.

Host side debugging follows standard practices for C++ projects. Host projects that use an IDE will follow the best practices recommended by that IDE. Projects that are written and run in Linux terminals can use a debugger like GNU Project Debugger (GDB). This is true for both low and high level design approaches.

D. Binary Neural Networks

FPGA has recently been at the forefront of neural network acceleration. With Arria 10's hardened floating point and reconfigurable fabric, FPGAs can now provide a wide range of options for various neural network architectures and configurations including single precision inference and binary acceleration [7][17]. Various methods have been explored for accelerating networks on FPGA but not until recently have these been competitive. Common problems seen in previous attempts are found in compute cycles per byte which leads to many designs being bandwidth limited [12]. This is especially seen in designs that focus only on convolution layers and not entire networks. Methodologies that attempt to overcome this are found by exploring roofline models [16] to balance between compute and external memory accesses, or by placing the entire network on-chip, but still do not reach state of the art performance. DLA [17] is one of the first designs to overcome many of the issues plaguing FPGA implementations by introducing a method to minimize bandwidth through on-chip caching and batching. The authors also contribute the first application of Winograd transform for FPGA to reduce the number of operations needed by multiply accumulates.

Binary networks were first employed to quantize filter data down to 1 bit [9]. Until recently, few designs have been published on accelerating binary networks [8][12][13] and even fewer prototyped and benchmarked. In the following section, we provide our experience into the development of creating an accelerator for binary weighted networks for the MCP platform.

III. BINARY WEIGHTED NUERAL NETWORK ACCELERATOR (BWNA) CASE STUDY

A. Method and motivation for accelerating CNN

Convolution Neural Network (CNN) layers generally take ~90% of the overall execution time. Thus, we aim to accelerate the CNN layers to speed up the application.

Various methods for accelerating convolution layers exist in the form of dot product engines, FFT, and matrix multiplication. Frameworks like Caffe [10] take advantage of highly optimized linear algebra libraries for CPU such as MKL BLAS for implementing these methods. CPUs handle these types of problems well because of their long history with compiler optimizations, caching abilities, and general yet flexible instruction sets. FPGA projects that attempt to accelerate neural networks in this manner tend to be bottlenecked by bandwidth limitations and seek various techniques to compete. Since the Intel Xeon CPU with

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

integrated FPGA provides a unique and innovative platform capable of handling these workloads we choose to accelerate our convolutions by formulating the problem as a matrix multiplication.

VGG is a very deep convolutional neural network specifically designed to increase the accuracy of image classification [11] using the ImageNet dataset [12][11]. Correlation between a networks depth and classification accuracy is shown to be positive. The authors in [11] outline 6 different architectures, 4 of which vary in depth from 11 weight layers to 19. We choose the architecture that includes 13 convolution layers for our experiments. In addition, we introduce batch normalization after every convolution and fully connected layer as it is shown to accelerate training and reduce the impact of the weights scale [15]. All non-convolution layers are executed on the Xeon processor in standard while in some cases convolution layers are offloaded specialized binary hardware on the FPGA.

B. BWNA Software Design

Caffe is a machine learning framework written in C++ that we used as the foundation for our accelerator framework. Changes to Caffe to support our application included enhancing certain functions for binarizing weights, integration of OPAE which in turn can add FPGA as a resource, and flexibility for offloading specific layers during network execution.

OPAE is interfaced with Caffe through a shared library that is responsible for receiving input and output pointers from Caffe, processing input data for the accelerator, and handling output data from the accelerator back to Caffe. During runtime, an instance of OPAE is created that allocates the required service and resources for that service.

C. BWNA AFU Design

We use a 2-dimensional systolic array architecture for mapping convolutional layers to BWNA (Fig. 5). This grid of processing elements (PE) is designed to perform arithmetic instructions as both filter and feature data are clocked in by peripheral feeder elements. To accelerate convolution layers we transform the problem into a binary matrix multiplication (bGEMM).

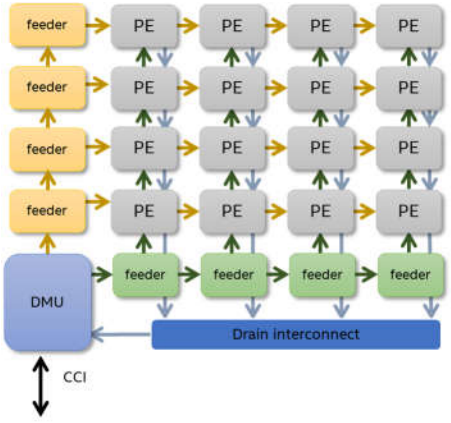

Fig. 5. BWNA bGEMM Accelerator

1) Blocking

Input matrices are generally too large to fit in the systolic array, so mapping larger matrices is done through blocking. Blocking breaks the problem into smaller dot products that are accumulated to produce the final output matrix. Matrix A is blocked by the number of PEs in the first dimension and matrix B is blocked by the number of PEs in the second dimension. Each matrix has a common block dimension derived from the dot product size. Blocking introduces flexibility into the systolic array design by offering tunable parameters for mapping different problem sizes.

2) Processing Elements

Each PE in our array implements a deterministic binarization function that contains specific logic that takes advantage of reducing a multiply accumulate to accumulate only. In application, multiplies turn into sign flips. Since hardware implementations of multiplies are significantly more expensive than adders, this PE function can be efficiently designed in core logic without the need for DSP blocks, leaving room for future DSP optimizations.

3) Data Conversion and Preparation

Due to network resilience, we choose BWNA to use 16-bit fixed data for bGEMM. Caffe only supports single and double precision data formats; this means for a CNN matrix multiplication the input matrices contain float data and the output matrix must contain float data. Fractional length is chosen such that there is minimal error when converting from floating point representation. Given a word length, WL = IL+FL where IL is the integer length, we define an imaginary point such that our float conversion is represented as IL.FL. Determining an optimal value of FL started with realizing our convolution outputs are processed through a batch normalization layer which limits the numerical range of the data. Combining this with the head room of input noise resilience, we concluded through experimentation that values for FL ranging from ~6-14 resulted in no loss in accuracy. Another approach takes the form of determining the largest integer value of the output activations across validation data sets and setting FL equal to the remaining number of bits; more formal, comprehensive methods can be found in [18][19].

D. BWNA Profiliing

AFU designs are profiled similarly to discrete applications in that there are no MCP specific profiling tools. Developers can add OS timers to OPAE code to measure AFU execution time. For additional AFU profiling, counters can be added inside AFU designs. Host code can be profiled using software such as VTune, or simple OS specific timers can be implemented over desired code blocks. The main performance metric we were interested in for BWNA was total execution time so we used standard operating system timers.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

IV. EVALUATION

A. Experimental Setup

Our overall goal is to determine a fine-grained offload scheme that produces the fastest execution time. We benchmarked our application for MCP comparing BWNA to the Xeon processor alone. From testing, we want to note which convolution layers BWNA executes faster than the Xeon, then offload those layers. We start by using our analytical model to find a series of systolic array architectures and choose the one that best fits our application. TABLE II. provides the systolic array parameters we used for our design. This design led to an AFU logic utilization of 28%. Rows and Cols define the PE matrix such that there are Rows*Cols PEs. The interleave factor is a parameter that controls data reuse lessening the burden on bandwidth. Dot is the size of the dot products within the PE. Giga Operations per Second (GOPS) is the peak performance achievable by the architecture and Avg. Effective GOPS is the average performance expected from the accelerator for a given efficiency mapping.

We choose to implement our design in HDL namely because of module reuse. Beyond the debugging topics covered in section II.C, we also used Verilog test benches as a method for unit testing. As a starting point for host code development, we create a simple framework for generating unit tests. Together, we verify the host code and systolic design using ASE. Once we have a fully compiled design we PR the bitstream on the FPGA and test the hardware with the same unit tests to ensure continuity between software verification and hardware implementation. Part of these unit tests were running perfectly mapped bGEMMS to determine if we could reach peak GOPS. Our tests concluded a peak measured TOPS of 2.19 out of a theoretical max of 2.2 TOPS. Next, we run the VGG network using our modified Caffe framework first on the Xeon processor then using BWNA. From these results, we determine which layers run the fastest on which architecture. Finally, we run the network using fine-grained offload on the layers than ran fastest on BWNA.

B. Experimental Results

TABLE III. shows the results of our experiments. BDX provides a faster total time, however, looking into each layer we see that BWNA executes faster in certain layers. BWNA has better potential performance and we note the correlation between operation efficiency (EFI), i.e., what percentage of operations performed contribute to output, and the layers that execute slower than BDX. Also note that conv1 is always run on the Xeon processor even when using BWNA because this layer takes an image as the input and the accelerator expects a normalized input. Our next experiment takes advantage of our framework and platform providing fine-grained offload and we choose to run the best performing layers on their respective architecture. This results in layers' conv5, conv6, conv7, conv8, conv9, and conv10 running on BWNA and the rest on the Xeon processor. These results can also be found in TABLE III.

TABLE II. SYSTOLIC ARRAY PARAMETERS USED IN EXPERIMENTS

Rows	16
Cols	16
Interleave	32
Dot	16
Peak GOPS	2211
Avg. Effective GOPS	960

TABLE III. EXPERIMENTAL RESULTS

Layer	BDX (ms)	FPGA (ms)	BWNA (ms)	Speed up	EFI (%)
conv1	382	390	386	1.0	-
conv2	2,066	6,074	2,147	1.0	9
conv3	878	1,626	890	1.0	18
conv4	1,772	2,397	1,804	1.0	22
conv5	810	688	689	1.2	39
conv6	1,619	1,126	1,125	1.4	44
conv7	1,619	1,123	1,127	1.4	44
conv8	792	432	432	1.8	77
conv9	1,577	792	792	2.0	77
conv10	1,577	792	765	2.1	77
conv11	409	489	426	1.0	38
conv12	409	474	424	1.0	38
conv13	409	504	421	1.0	38

C. Future Work

Because of limitations in bandwidth we were unable to test certain models for systolic array configuration. It is possible to achieve high (~90%) overall efficiency mapping but given the nature of these configurations, bandwidth requirements are too demanding for the current AFU architecture. Future updates to the systolic architecture will aim to reduce the bandwidth requirements. To start, we will improve storage efficiency by packing multiple weights into a single word. Smarter feeders will be designed to know how long a block of data needs to remain on chip before fetching new data, this will cut the required bandwidth down significantly. We plan to incorporate a streaming interface to improve data management to reduce the overhead of fine-grained offload. Currently, we are implementing convolution layers but we plan to extend our design to include support for inner product layers and training.

V. CONCLUSIONS

We introduced our journey in developing an application for Intel's Xeon processor with integrated FPGA and provided motivation for future work. We showed how the platform is capable of accelerating fine-grained workload style problems through the design of a binary weighted network accelerator. Utilizing the low latency, cache coherent QPI link, we could accelerate individual convolution layers in some cases 2x more than traditional standalone CPU solutions.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

REFERENCES

[1] Intel Xeon E5 Brief. http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-brief.html

[2] Altera Arria 10 Website. https://www.altera.com/products/fpga/arriaseries/arria10/overview.html

[3] Altera, 2014. The Industry's First Floating-Point FPGA. https://www.altera.com/en_US/pdfs/literature/po/bg-floating-point-fpga.pdf

[4] Intel. An Introduction to the Intel® QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[5] Intel Online Support for Quartus Prime. https://www.altera.com/products/design-software/fpga-design/quartus-prime/support.html

[6] Intel FPGA SDK for OpenCL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[7] E. Nurvitadhi, et al., "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?", ISFPGA, 2017.

[8] G. Venkatesh, E. Nurvitadhi, D. Marr, ".Accelerating Deep Convolutional Networks Using Low-Precision and Sparsity," ICASSP, 2017.

[9] M. Courbariaux, Y. Bengio, J-P. David "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," NIPS 2015.

[10] Y. Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding", 2014. http://ucb-icsi-vision-group.github.io/caffe-paper/caffe.pdf

[11] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. CoRR, abs/1409.1556.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09, 2009.

[13] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN:An ultra-low power convolutional neural network accelerator based on binary weights. CoRR, abs/1606.05487, 2016.

[14] Y. Umuroglu, et al. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. ISCAS, 2017.

[15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In arXiv:1502.03167, 2015

[16] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks.

[17] U. Aydonat, et al. An OpenCL Deep Learning Accelerator on Arria 10.

[18] Philipp Gysel, "Ristretto: Hardware-oriented approximation of convolutional neural networks," arXiv:1605.06402, 2016.

[19] Gupta, Suyog, Agrawal, Ankur, Gopalakrishnan, Kailash, and Narayanan, Pritish. Deep learning with limited numerical precision. arXiv preprint arXiv:1502.02551, 2015.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE