Accelerating HotSpots in Deep Neural Networks on a CAPI-based FPGA

Md Syadus Sefat
Virginia Tech
Blacksburg, VA
Email: sefat@vt.edu

Semih Aslan
Texas State University
San Marcos, TX
Email: aslan@txstate.edu

Jeffrey W Kellington
IBM Corporation
Austin, TX
Email: jwkellin@us.ibm.com

Apan Qasem
Texas State University
San Marcos, TX
Email: apan@txstate.edu

***Abstract*—This paper introduces a new energy-efficient FPGA accelerator targeting the hotspots in Deep Neural Network (DNN) applications. Our design leverages the Coherent Accelerator Processor Interface (CAPI) which provides a coherent view of system memory to attached accelerators. Our implementation bypasses the need for device driver code and significantly reduces the communication and I/O overhead. Performance is further improved by a tiling transformation that exploits data locality in the computation kernel via the CAPI Power Service Layer (PSL) cache. A new adder tree configuration is proposed which achieves a tunable balance between resource utilization and power consumption. An implementation on a CAPI-supported Kintex FPGA board achieves up to 155 GOPs/s and 15.79 GOPs/watt, improving on the state-of-the-art of FPGA-based DNN implementations.**

1. Introduction

We have witnessed an exponential growth in digital data over the last decade. To extract knowledge from this sea of information, researchers have developed sophisticated data analytics techniques driven by Machine Learning (ML) algorithms. Deep neural networks (DNNs) play a central role in solving real-time tasks in various ML applications such as image detection and classification, face recognition, natural language processing, fraud detection, targeted marketing, autonomous driving, and financial forecasting [1], [2], [3], [4]. Because of the ability to train and classify data with high accuracy, DNNs have emerged as the *de facto* algorithm for data analytics frameworks and have received enthusiastic interest from both academia and industry alike [5], [6], [7].

Because of the extremely high demands for both data and computation, analytics workloads, at least in the *training phase*, are deployed on large-scale systems on data warehouses. To achieve improved energy efficiency, the power-hungry CPUs on these systems are complemented with accelerators such as GPUs, and ASICs. Performance-critical tasks are off-loaded to the accelerators, which work in concert with the CPUs to attempt to deliver the desired performance at acceptable energy levels [8], [9]. In this context, re-configurable FPGAs offer an attractive alternative in accelerating DNN algorithms [10], [11], [12], [13]. FPGAs can deliver higher performance per watt compared to GPUs [14] and are more cost-effective than ASICs [15], striking the right balance between these two classes of devices. Furthermore, FPGAs can be useful in the *inference phase* since inference tasks are often run on battery-powered edge devices [16].

This work was supported by the National Science Foundation through awards CNS-1253292 and OAC-1829644 and by an equipment grant by IBM Corporation

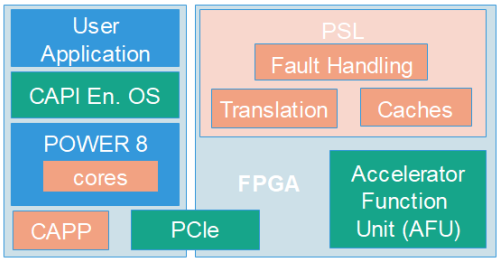

Figure 1. POWER8 CAPI Architecture

Hardware acceleration of DNNs poses significant challenges, however. Real-world DNNs are constructed with millions of model parameters requiring hundreds of megabytes of storage for *each* layer, which far exceeds the capabilities of current FPGAs. As a result, accelerated implementations incur high I/O overhead and performance is often dominated by data transfer time over a low-bandwidth I/O bus such as PCIe. Another major obstacle with FPGA acceleration, DNN or otherwise, is the time to development. In traditional HW-SW collaboration paradigm, the accelerator is attached as a memory-mapped I/O device. A device driver performs the virtual to physical address translation and delivers the addresses of the pinned kernel buffer to the accelerator. The burden of writing a custom device driver falls on the developer.

This paper describes a new flexible approach to implementing energy-efficient DNNs on FPGAs. Our implementation utilizes the Coherence Accelerator Processor Interface (CAPI), recently introduced by IBM [17] (Fig. 1). CAPI-supported systems allow the CPU and the attached accelerator to share the same coherent memory space. This feature provides two significant advantages.

(i) Coherent access to system memory implies that data does not need to be explicitly copied to device memory

prior to launching the computation on the FPGA board. This opens up new avenues for implementations that can mitigate the capacity and bandwidth constraints faced by FPGAs when executing DNN kernels.

(ii) Pointers within the shared address space can be passed freely between the CPU and the FPGA. This obviates the need for writing a custom device driver to access memory-mapped I/O and communicate with the accelerator, which can significantly reduce development time.

We exploit the above CAPI features and present a new pipelined implementation for the matrix-multiplication kernel within DNN applications. Matrix-multiplication is at the core of neural network algorithms and is considered to be the *principal hotspot*. It has been shown, that in a typical DNN application up to 95% of the time is spent performing matrix multiplication [18], [19], making it an ideal candidate for accelerator off-load. The key idea in our FPGA design is to fetch the matrix data from system memory in pipelined fashion and instantiate hardware blocks for a single row rather than for each data point in the output matrix. The hardware blocks are reused during the computation of each row and are populated with data for the next row when calculation for one row is complete. We augment this design with a tiling transformation that exploits data locality within the pre-fetched data in device memory. We implement tiling for both resource-constrained and unconstrained systems. We further optimize the design for resource usage by developing a novel tree-structured configuration for the adder units.

We also propose a new design for non-linear activation functions within DNN applications. Specifically, we look at the Rectified Linear Unit, which is the most commonly used activation function in deep learning. We provide the full implementation on a CAPI-supported Xilinx Kintex FPGA board. Experimental evaluation shows that the proposed design yields improved performance/watt over prior efforts. Our evaluation also shows that although DNN applications are dominated by matrix-multiplication computation, optimization of the activation layer tasks can also have a significant performance impact. To summarize, the main contributions of this work are as follows:

- a pipelined implementation of the matrix-multiply kernel in DNNs on a coherently attached FPGA
- a method that leverages CAPI features to exploit data locality in device memory
- a new tree-structured layout for adder units for improved resource utilization
- an FPGA implementation of the Rectified Linear Unit, which makes the case for accelerator off-loading of other layers in DNN application

2. Background

2.1. CAPI

The CAPI technology, introduced by IBM in 2014, is a hardware-software interface that enables coherent connection to custom acceleration engines within a heterogeneous compute unit [17]. Fig. 1 gives an overview of the CAPI architecture. The Coherent Accelerator Processor Proxy (CAPP), hosted on POWER CPUs and the Power Service Layer (PSL), installed on the accelerator, work collaboratively to provide a coherent view of system memory. This essentially turns the accelerator into a functional unit within the CPU (called Accelerator Function Unit or AFU in CAPI terminology). The PSL contains a 256 KB cache which can be accessed by the accelerator. An application running on the CPU sends a *Work Element Descriptor (WED)* to the AFU. WED contains pointers to data and meta-data in the system userspace on which accelerated computation is performed by the AFU. This is in stark contrast to the traditional paradigm, in which the accelerator is attached as a memory-mapped I/O device, and device driver performs the virtual to physical address translation.

CAPI adds an Effective-to-Real-Address Translator (ERAT) within PSL which translates the addresses, significantly reducing address translation overhead. In traditional systems, when an accelerator finishes its computation, it writes the data in the kernel space. A device driver then copies the data to the user space, generates a pointer to the data, and passes it to the application. Thus, the same data is copied *twice*. In CAPI, the pointer to the user space data is sent from the application directly to the FPGA, thereby eliminating the cost of copying data from kernel to user space.

2.2. Deep Neural Networks

A neural network composed of more than one hidden layer qualifies as a *deep* neural network. In practice, however, DNNs can consist of many layers containing millions of nodes. Each layer trains on a distinct feature vector based on the previous layer's output. Each layer is initialized with a *weight matrix* and then goes through multiple forward and backward passes. In forward propagation, a *loss function* is evaluated and output is calculated from each layer which is fed into the next layer. In backward propagation, gradients of the parameter matrices are calculated (e.g., Jacobian matrix) and weight matrices in different hidden layers are updated iteratively to minimize the loss function. In general, in a forward-pass, a huge amount of data is cached in memory, and during back-propagation, those values are reused for updating the loss function. Each node of any layer is activated by an *activation function* such as Sigmoid, or Rectified Linear Unit (ReLu). After being triggered by the activation function, the node produces an output.

Fig. 2 shows the workflow of a typical DNN in terms of the core computation tasks. The figure also shows the mapping of the tasks to the available computation units. The fully connected layers which involve matrix computation are the prime targets for offloading to an FPGA. In this work, we show that the activation functions (highlighted in red rectangles) are also suitable for acceleration.

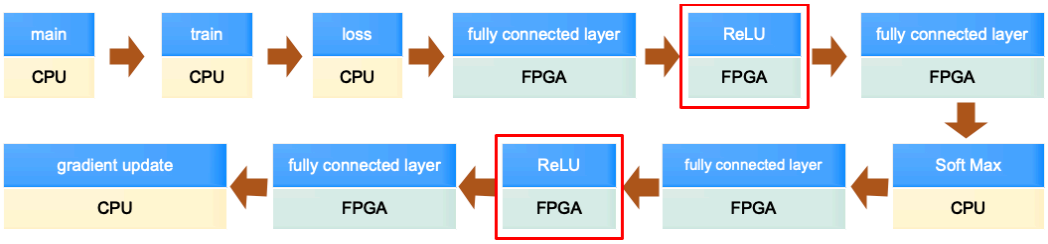

Figure 2. DNN workflow and their mapping to computation units in a CPU-FPGA collaborative environment

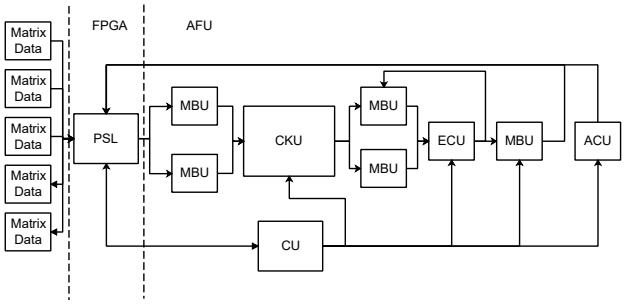

Figure 3. FPGA hardware architecture overview

3. Hardware Design

The design goal of our FPGA is to take advantage of the shared coherent space afforded by CAPI. The matrix-multiplication operation is decomposed into a series of vector dot-product computations, and a specialized hardware unit is constructed to execute the dot products in parallel.

3.1. Architecture overview

This section represents an overview of the architecture. Fig.3 depicts the hardware architecture. The AFU consists of computational kernel units (CKUs), a controller unit (CU), multi-ported buffer unit (MBU), extra computational unit (ECU) and an address computing unit (ACU). The CKU consists of two specialized computation kernels (i) matrix-multiplication (MMU) and (ii) ReLu unit (RLU). MMUs are the largest computational blocks in the accelerator. An $n \times n$ MMU kernel consists of n vector-dot-product (VDP) units. Each VDP unit computes the vector dot product of two vectors of size n. In the baseline design, a VDP unit consists of n multipliers and $(n/2)$ adders. The CU is responsible for extracting information from meta-data in WED and maintaining hardware state. The ECU is used in the pipelined computation for performing additions on intermediate data. The ACU calculates the virtual address for reading and writing user space data to- and from- the coherent shared memory.

We first describe a non-CAPI based canonical implementation of matrix-multiply in this architecture and then present our pipelined implementation.

3.2. Canonical implementation

In an $A \times B = C$ matrix multiplication, elements of the i^{th} row in A are combined with elements of the j^{th} column of B to form the resultant $C(i,j)$ element. An element-wise multiplication is performed between $A(i:)$ and $B(:j)$ followed by a summation of results of the multiplication. In the canonical implementation, multipliers and adders are dedicated to computing each element in the resultant matrix C. For an $n \times n$ weight matrix, to compute each element in the resultant matrix, n multipliers are initialized which operate in parallel. The summation cannot be fully parallelized due to dependencies and is therefore performed in stages. For each element of the resultant matrix, a total of $n/2$ adders are initialized. In the first stage, $n/2$ additions are completed in parallel. Each subsequent stage operates on the results from the previous, thus reducing the number of additions by half. Adders are re-used from one stage to the next via a buffering mechanism. For $n/2$ adders, number of stages required to complete the summation is $log_2(n+1)$.

In the first stage, input to the adders, $\{m_1, m_2, \ldots m_{16}\}$ are received from the output of the multipliers. The adders produce output $\{s_1, s_2 \ldots s_8\}$ which are propagated on to the next stage. In this design, the number of required adders and multipliers grows quadratically with the problem size. For an $(n \times n)$ matrix-multiplication with canonical design, a total of $(n \times n \times n)$ multipliers and $(n \times n \times n/2)$ adders are required. This can prove problematic when n is large, as larger instances will fail to execute on many of today's FPGAs due to resource limitations. Furthermore, a non-CAPI implementation will require the entire A and B matrices to reside in the on-chip buffer of the FPGA before the computation can begin. This will cause further strain on the limited accelerator resources.

3.3. CAPI-based Pipelined Implementation

Our enhanced CAPI-based design exploits pipelined parallelism and access to shared memory space and can operate with limited resources while still delivering superior performance. The key idea is to instantiate hardware blocks for a single row rather than for each data point in the output matrix. The hardware blocks are reused during the computation of each row, and data is passed to the blocks in a pipelined fashion when computation for one row is complete. An additional benefit of this approach is that it is

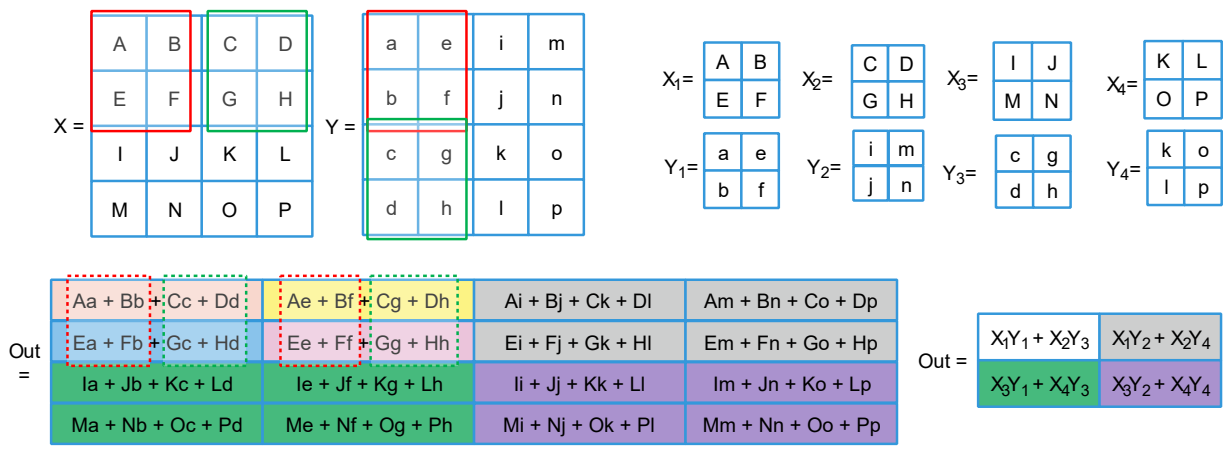

Figure 4. Tiled computation of matrix-multiply on coherent FPGA

only necessary to keep one multiplicand in device memory at a time. This allows for better utilization of bus bandwidth (PCIe) between host and device.

The design consists of an MMU unit which contains all computational hardware blocks. For an $n \times n$ matrix multiplication, MMU instantiates n hardware blocks with each block containing n multipliers, $n/2$ adders and two data buffers, D_A and D_B of size n. The two buffers store the row and column elements of the two input matrices, respectively. To begin the computation, $A(0:)$ is copied into the D_A buffer in each hardware block, while $B(:j)$ (the j^{th} column in B) is copied into the D_B buffer in the j^{th} hardware block. Each hardware block then performs n multiplications in parallel and $n-1$ additions in $log_2(n/2)$ stages to produce one element in $C(0:)$. The hardware blocks themselves execute in parallel, and thus an entire row of C is computed in parallel. The result is buffered, and its transfer to the physical storage location of C is overlapped with subsequent computation. In the next stage, $A(1:)$ is copied into D_A overwriting $A(0:)$ and the same steps are repeated. Note, D_B remains untouched after the first iteration, as the entire B matrix is fetched during the initial stage.

4. Design Optimizations

4.1. Tiled Computation

We improve on our design described in Section 3 by developing a tiled-version of the matrix-multiplication unit. This design exposes temporal locality which we exploit via the CAPI PSL cache. This design also further reduces the storage requirements on the FPGA, enabling us to tackle larger matrices, representative of real data sets.

In the tiled design, larger weight matrices are segmented into smaller sub-matrices. Each MMU kernel operates on these smaller sub-matrices, storing the intermediate results in temporary buffers until the final result is computed. The model exhibits high degrees of temporal locality since each sub-matrix is accessed many times by the accelerator with the degree of reuse increasing with matrix size.

Fig. 4 illustrates our tiling strategy. Consider two 4×4 matrices, *X* and *Y*. The first contains the letters *A-P* and the second *a-p*. Each matrix is segmented into four 2×2 sub-matrices. X is tiled into $X_1 - X_4$ while Y is tiled into $Y_1 - Y_4$. The result of the matrix multiplication between *X* and *Y* is shown in the *Out* matrix in expression form. We observe that if we group the first two elements in the $Out(1,1)$ expression with the first two elements in $Out(2,1)$, it produces the result of multiplying X_1 and Y_1. Similarly, if we group the second two elements in $Out(1,1)$ expression with the first two in $Out(2,2)$, it produces $X_2 \times Y_3$. Finally, $X_1Y_1 + X_2Y_3$ gives a 2×2 matrix whose elements are exactly the same as the elements of the first quadrant of the matrix *Out*. The remaining quadrants can be computed in a similar fashion.

Implementation of tiled matrix-multiply requires minimal changes to the baseline architecture presented in Fig. 3. Additional data buffers (MBUs) are needed to hold the input sub-matrices, intermediate results, and output sub-matrices. These MBUs are implemented as BRAMs on the board. CKUs are replicated to process each sub-matrix in parallel. After computing the intermediate matrix multiplication, data is passed to the EKU for element-wise multiplication among the intermediate matrices.

4.1.1. Tiling for resource-constrained systems. Depending on the availability of resources in an FPGA, the tiled matrix multiplication can be designed in two ways. In the first approach, all intermediate result matrices are buffered locally within the FPGA. The second approach assumes the FPGA does not have sufficient BRAM to buffer all intermediate results. After computing the intermediate matrix multiplication in the CKU (Fig. 3), the data is passed to the EKU for element-wise multiplication. Depending on the requirements of the addition stages, the CU requests input data from system memory via the PSL. When the addition

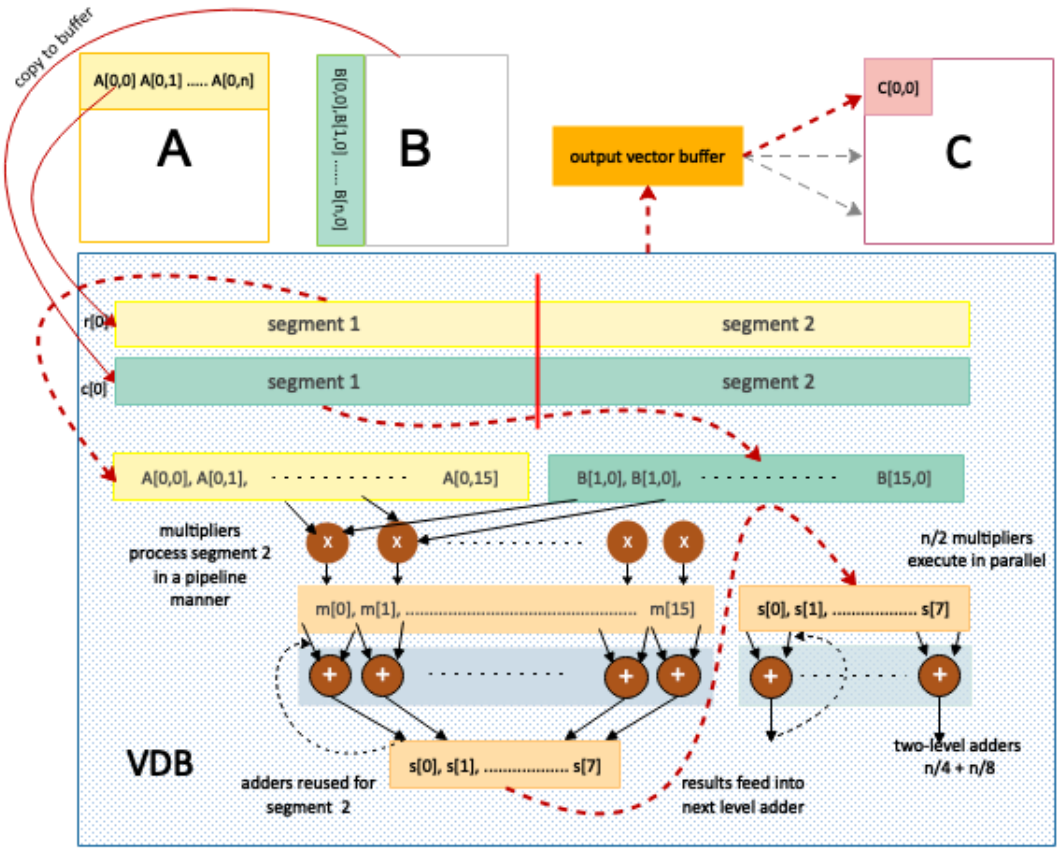

Figure 5. Resource-conscious MMU unit (MMU_RC) with limited-depth adder tree. Figures shows data flow for computing C(0,0) in a single VDB unit

is completed, the control unit directs the PSL to write back the result into system memory. As the number of MBUs is limited in the design consideration, the AFU may *not* be able to use the same input tile multiple times and the PSL may need to fetch the same data several times. Therefore, this resource-aware implementation will come at a cost of the reduced locality.

4.1.2. Selecting a Tile Size. The tile size can have a significant impact on the performance of our implementation. Selecting a tile that is too large will lead to an unexploited locality in the PSL cache while a smaller tile size can cause extra overhead. We take a conservative approach in selecting the appropriate tile size. We assume each FPGA has a computational unit that can compute a matrix multiplication with a *minimum* size. We call this the *kernel* and set this as the *maximum* size for each sub-matrix. We base this selection on the following rationale: selecting a smaller tile size will maximize reuse in the PSL cache (even for caches with smaller capacity), but at the same time it will enable concurrent processing of the maximum number of tiles allowed by the system.

Let, $kernel = n \times n$, size of the input matrices $= N \times N$ and $k = N/n$. Then the total number of tiles (and sub-matrices) is given by $(N \times N)/(n \times n) \times 2 = 2k^2$; total number of (sub) matrix-multiplications is given by k^3 and the number of additions required is $(k-1)k^2$. Since the number of matrix multiplication is k^3, we need k^3 buffers to hold the intermediate results and another k^2 to hold the final results. So the total number of buffers required is $3k^2 + k^3$. We can use this as a basis to compute the largest matrix sizes that can be efficiently processed by our design on a particular board. For example, The KU115 FPGA has 4.5KB 2160 BRAM blocks. If we consider 32×32 single-precision floating-point matrices, a tiled matrix will require 4KB space. Hence, using our design, without re-utilizing the MBUs, we can efficiently compute a 352×352 matrix-multiplication with $k = 11$.

4.2. Activation Functions

To improve the energy efficiency of the overall application, in addition to matrix-multiplication, we developed an FPGA implementation for a linear activation function. Specifically, we focused on the Rectified Linear Unit (ReLU), which has become the *de facto* choice in recent years due to its simplicity and ability to enable fast training [20]. ReLUs in DNNs compute the function $f(x) = max(0, x)$ which triggers the neurons in the previous layer. We designed a hardware computation unit, RLU to perform

this operation. The RLU checks the sign bit of the input operand and passes it to the output if the sign bit is 0 otherwise it replaces the input data with 32-bit zeros. Thus it accomplishes the functionality of a basic ReLU operation. ReLU unit takes only one clock cycle to complete its operation. A 32×32 ReLU unit with 150 MHz clock performs 256 GFLOPS in an FPGA.

Generally, non-linear activation layers are placed after each fully connected layer. If the activation layers are implemented in the host side then there is a need to move data from FPGA to the host. Due to the high latency of PCIe, it may take several cycles to transfer a single byte of data. Since the FPGA implementation of the activation function takes only one cycle to operate, the entire cost of this data movement can be eliminated.

4.3. Resource-conscious Convolution

We modified the MMU computation block to make it more resource-conscious. This new MMU, referred to as MMU_RC, optimizes the area utilization on the FPGA to reduce power consumption. In the MMU_RC, the input matrices are divided into two segments and the computation on the two segments are performed using pipelined-parallelism. Hardware resources are reused in each stage of the pipeline.

Fig. 5 shows the basic configuration of the resource-conscious multiplier. For an $n \times n$ matrix multiplication, The VDB unit in MMU_RC consists of one row of $(n/2)$ parallel multipliers. The adder units are configured in a reverse-tree organization. The first level of the tree includes of $n/4$ adders, and the number of adders is cut in half for each subsequent level. Adder trees in a given level run in parallel. The depth of the tree is tuned for resource utilization. For maximum utilization, a complete tree of depth $log_2(n/2)$ is constructed while for minimum utilization and maximum power savings, the adders are arranged in a single row of $n/4$ adders. The optimal trade-off point lies somewhere in between. Fig. 5 shows an arrangement with a two-level tree consisting of a total of $(3n/8)$ adders. As in the baseline MMU, MMU_RC also contains two buffers to hold an *entire* row and column of the two input matrices, respectively.

When MMU_RC is initiated, the first segment of the input data is passed to the multiplier row. After completion of the multiplications on the first segment of data, the result is transferred to the first level of adders, ADD_1 as input and the result from ADD_1 is sent to ADD_2 as input. In parallel to the additional operations, the second block of input data is transferred to the input buffer of the multipliers. When results from the first stage have been sent from ADD_1 to ADD_2, ADD_1 becomes available for the second stage computation. The final result for a single row is available after two stages of multiplication and $log_2(n/4)+log_2(n/8)$ stages of addition.

TABLE 1. PLATFORM DETAILS

FPGA	Clk	250 MHz
	CLB LUTs	663360
	CLB Registers	1326720
	Block RAM Tile	2160
	PSL Cache	256 KB
	PCIe	Gen3
CPU	CLK	3.2 GHz
	RAM	256 GB
	L1 Cache	64 KB
	L2 Cache	512 LB

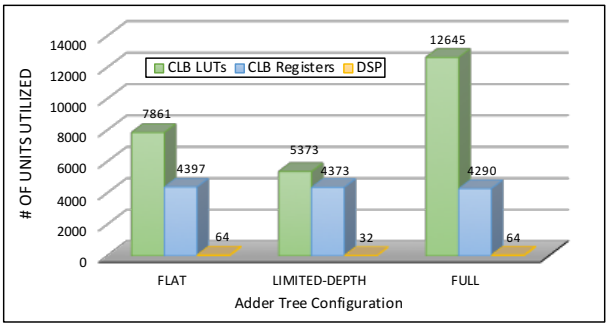

Figure 6. Resource utilization with different adder tree configuration

5. Evaluation

5.1. Experimental Setup

We evaluated our design on a CAPI-enabled POWER8 system which hosts a Xilinx Kintex Ultrascale 115 FPGA. Xilinx Kintex has a xcku-115-flva1517-2e chip on an Alpha Data ADM-PICE-KU115 board with an X16 PCI-Express interface. Table 1 provides further details.

We used Xilinx Vivado's Out of Context (OOC) synthesis and implementation feature to synthesize and route results. OOC enabled us to synthesize the RTL design for which the pin numbers exceed the maximum pin numbers of the FPGA board. For offloading, implementing, routing, and placement of the architecture, non-project batch mode of design flow has been used with *tcl* scripting. To simulate the software-hardware interaction in CAPI, PSLSE simulator was used[21]. For each experiment, we instantiated the application with randomly generated weights and dimensions for the fully connected layers. We generated *reference output* using a reference CPU implementation. We validated our implementation against this reference output. Each performance experiment was repeated 10 times, and only the average numbers are reported here.

5.2. Resource Utilization

The primary metric to quantify the capacity in an FPGA is CLB utilization. CLB utilization is an important parameter as it plays a vital role in the complexity of the circuit when it is placed and routed in the FPGA. For our design, the CLB utilization is a function of the computational units and the data buffers in the architecture. Fig. 6 reports resource usage

TABLE 2. POWER CONSUMPTION ESTIMATION OF ON-CHIP COMPONENTS

On-Chip Components	Power (W)	% power
Clocks	0.606	4.06
CLB Logic	1.427	9.57
—–LUT as Logic	1.382	9.27
—–Register	0.031	< 1
—–CARRY8	0.013	< 1
—–F7/F8 Muxes	< 0.001	< 1
—–Others	0.000	0.00
Signals	1.034	6.93
DSPs	0.261	1.75
I/O	10.002	67.07
Static Power	1.583	10.61
Total	14.912	100

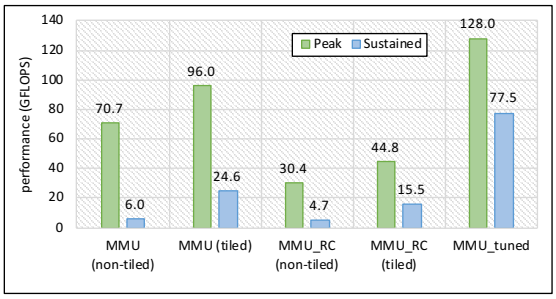

Figure 7. Peak performance and achieved sustained performance for different implementations

of the MMU_RC computation kernel in terms of CLB and DSP utilization. Data is shown for three different adder tree configurations: (i) flat tree (single row of adders) (ii) limited-depth (two rows) and (iii) full tree (depth = $log_2(n/4)$. We observe that the full tree configuration utilizes more CLBs than the other two. It utilizes the same number of DSP blocks as the flat tree but twice as many as the limited depth tree. Not much variation is observed in terms of CLB register usage. Overall, the limited depth tree is the best choice for the resource-constrained boards.

5.3. Power Analysis

We back-annotated the switching activity in Vivado's RTL simulation to estimate our design's power consumption. Vivado supports both dynamic and static power estimation. Dynamic power is estimated at an ambient temperature of 25°C and includes power dissipation due to input data access patterns and the design's internal activity, and total on-chip and off-chip power. Static power measures the power consumption as a result of transistor leakage current.

Table 2 shows the on-chip power consumption of the MMU computation kernel for a 32×32 matrix-multiplication. The report is generated from an OOC synthesized architecture. We observe that a large portion of the total dynamic power is consumed in the I/O bus. As it is an OOC synthesized model, all input and output port declarations are mapped onto the I/O pins. In a physical implementation, the unit will be instantiated in the PSL, and there will be no associated I/O power cost. As expected, the LUT logic activity dominates the rest of the dynamic power consumption. BRAM activity accounts for close to 30% of the off-chip dynamic power while PCIE accounts for a mere 7%. This indicates that our approach of pipelined matrix-multiply in which matrices are fetched in a staggered fashion over PCIe, does not have a significant negative impact on off-chip power consumption.

5.4. Performance

Fig.7 shows the theoretical peak and the actual achieved performance of the different design variants discussed in this paper. The theoretical peak is calculated as the sum of adders and multipliers multiplied by the maximum operating frequency. We look at the performance of both the pipelined (MMU) and the resource-conscious (MMU_RC) implementations. For each of these, we also consider the impact of tiling. Finally, we look at a fifth category that is tuned for both the tile size and the adder tree configuration.

We observe that tiling provides a significant performance boost for both MMU and MMU_RC, yielding integer factor improvements over the non-tiled counterparts. As expected, the resource saving mechanisms in MMU_RC causes some performance to be sacrificed. In general, the resource-oblivious implementations almost double the performance of MMU_RC. The fraction of peak achieved by the different design is somewhat low (30% average). Notwithstanding, these numbers are fairly typical for these types of designs, particularly when an attempt has been made to conserve power.

Finally, we observe that the tuned-MMU is a head above all other implementations. This reiterates the need for the careful selection of tile sizes and adder tree configurations. The power numbers presented in § 5.3 are obtained from the tuned implementation. When excluding I/O overhead, our tuned implementation can deliver 15.79 GFLOPs/watt for a 32×32 matrix multiplication.

5.5. Comparison to Previous Work

We compare our design with some recently proposed FPGA implementations of matrix multiply [22], [23], [24], [25]. Performance of the previous implementation as reported in [22], [23], [24], [25] are shown in Table 3. Our implementation not only achieves the highest performance but also has better energy efficiency. It should be noted, however, that none of the previous efforts targeted a coherently attached accelerator (i.e., CAPI). Also, currently, our design only supports 32-bit floating point data; theoretically a fixed-point implementation will demand fewer FPGA resources.

6. Related Work

In recent years, there has been a plethora of work in accelerating DNN applications. Most of these use GPUs or ASICs as the acceleration device. Work on FPGA-based

TABLE 3. COMPARISON WITH PREVIOUS WORK

	[22]	[23]	[24]	[25]	Ours
Year	2010	2014	2015	2016	2018
Platform	Virtex5 SX240t	Zynq XC7Z045	Virtex7 VX458t	Zynq XC7Z045	CAPI KU115
Clock (MHz)	120	150	100	150	250
Quantization	48-bit fixed	16-bit fixed	32-bit float	16-bit fixed	32-bit float
Performance (GOP/s)	16	23.18	61.62	136.97	155.08
Power(W)	14	8	18.61	9.63	9.82
Energy Efficiency (GOP/s/W)	1.14	2.90	3.31	14.22	15.80

acceleration is limited and have mostly focused on Convolution Neural Networks (CNN) rather than DNNs in general.

Chakradhar *et al.* present a dynamically configurable architecture for CNN. Their design exploits parallelism and optimized memory throughput across different layers. The techniques are implemented on-the-fly based on the demands of a specific layer and a particular CNN workload [22]. Gokhale *et al.* present an accelerator for DNNs targeting mobile computing platforms (rather than data centers). They use pipelined parallelism and quantization (Q8.8) as optimization methods. Their implementation achieves a peak performance of 200 GOPs/s [23]. Suda *et al.* explore the design space of OpenCL-based accelerators for CNNs. They identify key design variables and demonstrate that fine-tuning these parameters, both analytically and empirically, can lead to improved hardware acceleration [11]. Qiu *et al.* show that in general fully-connect layers in CNNs are memory bound while the convolution layers are computed bound [25]. The develop a dynamic-precision data quantization method and a convolver design to address these issues. Their CNN implementation on a Xilinx Zynq ZC706 board achieves 137 GOPs/s. Caffeine [12] and Dianno[26] frameworks both provide acceleration support for DNN workloads. However, these designs are meant to be used standalone with very lightweight processors to control the flow. The work presented in this paper is distinct from earlier work on DNN because none the previous efforts target a coherently attached FPGA, as we do.

There have been a few studies of non-DNN applications on CAPI-supported systems. For example, Giefers *et al.* accelerate FFT on a CAPI-enabled FPGA accelerator. They demonstrate that CAPI's shared virtual memory space can significantly reduce system call overheads for the FFT kernel [27]. Ito and Moriyoshi present an implementation of the Pair-HMM algorithm optimized with multi-threading and data padding on a POWER8-CAPI system [28]. To the best of our knowledge, this is the first attempt at optimizing DNN applications on a CAPI-based system.

7. Conclusions

This paper presented a CAPI-based accelerated implementation of key computation steps in a Deep Neural Network application. The work demonstrates that utilization of the shared virtual memory, as enabled by the CAPI technology, can provide a significant performance boost for these kernels. Specifically, we show that the coherent system memory space can be leveraged to optimize data transfer times between host and device and exploit temporal locality within the the matrix-multiply kernel. Our implementation not only yields superior performance than previous efforts but also delivers higher performance/watt. The results also show that the activation functions in DNN are also suitable candidates for FPGA acceleration and can lead to overall performance benefits. Finally, the adder tree configuration described in this paper can help achieve a tunable balance between resource utilization and power consumption.

In future work, we plan to explore a hybrid implementation of DNN in which the CPU and the accelerator collaborate in executing the computation kernel by leveraging the shared memory space. We are also looking at quantization of the of floating-point data elements to further improve the performance of our CAPI-enabled implementation.

References

[1] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *CVPR14*, 2014.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *CVPR15*, 2015.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR14*, 2014.

[4] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A parallel random forest algorithm for big data in a spark cloud computing environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 919–933, 2016.

[5] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[6] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-gpu training of convnets," *arXiv preprint arXiv:1312.5853*, 2013.

[7] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 2211–2212.

[8] E. Dufrechu, P. Ezzatti, E. S. Quintana-Ort, and A. Remn, "Accelerating the Lyapack library using GPUs," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1114–1124, Sep. 2013.

[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *ISCA16*, pp. 243–254.

[10] H. Sharma and et al, "From high-level deep neural models to FPGAs," in *MICRO*, Oct. 2016, pp. 1–12.

[11] N. Suda and et al, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," in *FPGA16*.

[12] C. Zhang and et al, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *ICCAD16*.

[13] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang, "Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 20, p. e3850, 2017.

[14] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *ACM FPGA17*. ACM, 2017, pp. 5–14.

[15] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic," in *FPL16*. IEEE, 2016.

[16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[17] J. Stuecheli and et al., "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, 2015.

[18] Y. Jia, "Learning semantic image representations at a large scale," Ph.D. dissertation, UC Berkeley, 2014.

[19] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2d and 3d cnns on fpga," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 97–106.

[20] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[21] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné *et al.*, "Parallel programming models for a multiprocessor soc platform applied to networking and multimedia," *IEEE VLSI06*, vol. 14, no. 7, pp. 667–680, 2006.

[22] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[23] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPR Workshops*, 2014, pp. 682–687.

[24] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *ACM FPGA16*. ACM, 2015, pp. 161–170.

[25] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *ACM FPGA16*.

[26] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[27] H. Giefers, R. Polig, and C. Hagleitner, "Accelerating Arithmetic Kernels with Coherent Attached FPGA Coprocessors," in *DATE15*, 2015.

[28] M. Ito and M. Ohara, "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm," in *COOL CHIPS XIX*, 2016.