# A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform

## A Deep Learning Case Study

Duncan J.M. Moss*, Srivatsan Krishnan†, Eriko Nurvitadhi†, Piotr Ratuszniak†‡, Chris Johnson†, Jaewoong Sim†, Asit Mishra†, Debbie Marr†, Suchit Subhaschandra† and Philip H.W. Leong*

†Intel Corporation, ‡Koszalin University of Technology,*The University of Sydney
duncan.moss@sydney.edu.au,{srivatsan.krishnan,suchit.subhaschandra}@intel.com

## ABSTRACT

General Matrix to Matrix multiplication (GEMM) is the cornerstone for a wide gamut of applications in high performance computing (HPC), scientific computing (SC) and more recently, deep learning. In this work, we present a customizable matrix multiplication framework for the Intel HARPv2 CPU+FPGA platform that includes support for both traditional single precision floating point and reduced precision workloads. Our framework supports arbitrary size GEMMs and consists of two parts: (1) a simple application programming interface (API) for easy configuration and integration into existing software and (2) a highly customizable hardware template. The API provides both compile and runtime options for controlling key aspects of the hardware template including dynamic precision switching; interleaving and block size control; and fused deep learning specific operations. The framework currently supports single precision floating point (FP32), 16, 8, 4 and 2 bit Integer and Fixed Point (INT16, INT8, INT4, INT2) and more exotic data types for deep learning workloads: INT16xTernary, INT8xTernary, BinaryxBinary.

We compare our implementation to the latest NVIDIA Pascal GPU and evaluate the performance benefits provided by optimizations built into the hardware template. Using three neural networks (AlexNet, VGGNet and ResNet) we illustrate that reduced precision representations such as binary achieve the best performance, and that the HARPv2 enables fine-grained partitioning of computations over both the Xeon and FPGA. We observe up to 50x improvement in execution time compared to single precision floating point, and that runtime configuration options can improve the efficiency of certain layers in AlexNet up to 4x, achieving an overall 1.3x improvement over the entire network.

**ACM Reference Format:**
Duncan J.M. Moss*, Srivatsan Krishnan†, Eriko Nurvitadhi†, Piotr Ratuszniak†‡, Chris Johnson†, Jaewoong Sim†, Asit Mishra†, Debbie Marr†, Suchit Subhaschandra† and Philip H.W. Leong* and †Intel Corporation, ‡Koszalin University of Technology,*The University of Sydney. 2018. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study. In *FPGA '18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174258


*FPGA '18, February 25–27, 2018, Monterey, CA, USA*

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174258

## 1 INTRODUCTION

High performance and scientific computing (HPC & SC) workloads rely on the basic linear algebra (BLAS) [1] subroutines to perform many of their most time intensive functions. BLAS libraries are optimized for high performance and consist of three separate levels with level 3 routines focused on matrix operations. The general matrix to matrix multiplication (GEMM) level 3 routine is arguably the most time intensive and widely used function in HPC and SC. Hence when designing an accelerator for these applications, targeting the GEMM routine often leads to the highest improvement in performance.

In the past, architectures for accelerating these HPC and SC algorithms have been developed for discrete FPGA or FPGAs with embedded soft/hard processors, with all functions handled by the FPGA fabric. Often resources which would be better allocated to accelerating specific bottlenecks, such as GEMM, need to be traded off to implement the other less compute intensive portions of the algorithm. Recently, heterogeneous CPU+FPGA platforms have been proposed as an alternative to discrete FPGAs. By close integration with a CPU, the FPGA's resources can be better utilized to optimize the most compute intensive parts of the application, while less FPGA friendly functions are handled by the CPU. The Intel HARPv2 [12] combines a 14 core Broadwell Xeon CPU and an Arria 10 GX1150 FPGA. It provides access to the standard x86 ecosystem, coherent access to the CPU's memory and high bandwidth to the FPGA. In contrast to embedded heterogeneous platforms, the Xeon is designed for high performance, giving the users flexibility to partition the heavy compute in a way which best suits the FPGA and CPU. By using a heterogeneous CPU+FPGA platform like HARPV2, the designer has freedom to explore architectures that advantageously exploit collaboration between the FPGA and CPU. Hence, better performance can be achieved via specialized high speed accelerators that focus on the computational bottleneck, and rely on a high performance CPU to handle the rest.

Deep learning is quickly becoming a disruptive technology with state-of-the-art accuracy shown in applications such as computer

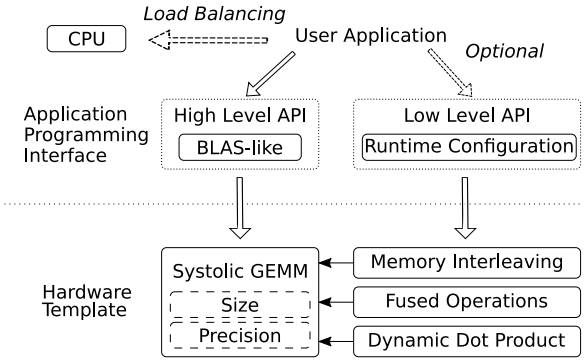
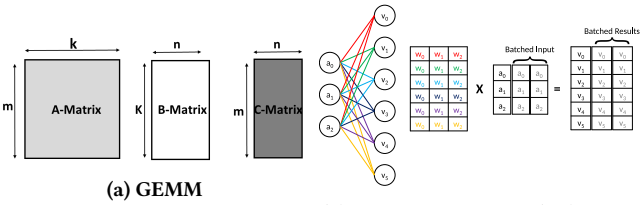



**Figure 1: The framework consists of an application programming interface and hardware template. The high level API provides a function call similar to those used in BLAS libraries. The low level API is optional and allows the developer to configure certain aspects of the hardware template at runtime.**

vision, autonomous driving, speech-to-text and artificial intelligence. It is still a very active area of research with work focusing on new topologies and reduced precision methods [3, 10, 17, 20, 21, 26, 26, 30]. While single precision floating point representation is still used in training, it has been shown that reduced precision representations all the way down to a single bit can be sufficient for inference. Due to lack of native low precision support in CPUs and GPUs, FPGAs are well positioned to take full advantage of exotic data types supported by their fine grained reconfigurable architecture. It has been shown [2, 4] that GEMM plays a significant role in deep learning.

This paper presents a customizable matrix multiplication framework on Intel's HARPv2, and an overview is shown in Fig. 1. It consists of a highly configurable hardware template with a streamlined software stack and runtime application programming interface (API), which allows for a wide range of different precisions, various core sizes and tunable runtime configurable parameters. We take deep learning as a case study to evaluate performance and flexibility of our framework. Various HW/SW co-design and heterogeneous load balancing techniques are applied to achieve synergistic collaboration between the Xeon CPUs and FPGA. Specifically, the contributions of this work are as follows:

- The first runtime configurable heterogeneous GEMM implementation which supports arbitrary matrix sizes and offers a wide range of precision, blocking, fusing of operations, buffering schemes and load balancing.
- A systolic GEMM template that allows runtime customization of memory interleaving, offering performance improvements of up to 2.7x on small matrices and 4x for certain neural networks. In addition, it incorporates a scheme for fusing operations so inline computation such as ReLU, Batch Norm and Clipping can be done in FPGA hardware, minimizing CPU overhead.
- A dynamic dot product, enabling mixed precision training and binary inference which leverages the HARPv2 architecture, providing up to a 1.67x improvement over a 14 core CPU.
- An evaluation of performance using popular deep neural networks (AlexNet, VGGNet and ResNet) on the HARPv2 platform used for ILSVRC15[23], and a study on the efficiency of the hardware template and its impact on deep learning performance. The resulting binary implementation is, to our knowledge, the fastest and most flexible reported to date.




**(a) GEMM** **(b) A Fully Connected (FC) Layer**

**Figure 2: (a) Matrix layout for a GEMM. m and n are the leading dimension and k is the common dimension. (b) Each neuron takes in activations ($a_i$) from the previous set of neurons and performs the dot product with its weights ($w_i$) to produce its output ($v_i$).**

The rest of this paper is organized as follows. Sec. 2 focuses on GEMM and neural networks. The API, hardware template and HARPv2 specific details are presented in Sec. 3 and Sec. 4. The fused operations and the dynamic dot product are presented in Sec. 5. The results are presented in Sec. 6 and related work is discussed in Sec. 7. Finally, conclusions are drawn in Sec. 8.

## 2 BACKGROUND

GEMM is a key function in many HPC and SC workloads and can be expressed as Equ. 1.

$$C = alpha * op(A) * op(B) + beta * C \quad (1)$$

Where $A$, $B$ and $C$ are the input and result matrices, $alpha$ and $beta$ are scaling factors and $op$ is a separate function that performs a matrix transpose if needed. The most computational intensive part of the operation is the $A * B$ matrix multiplication. As illustrated in Fig. 2a, matrices $A$ and $B$ share a dimension, $k$, named the common dimension and another unrestricted dimensions $m$ and $n$ named the leading dimensions. The output of the matrix multiplication is accumulated with the $C$ matrix and has dimension, $m$ and $n$. For each element in the result matrix a $k$-length dot product needs to be performed. Hence the number of multiply accumulates can be generalized to $m * n * k$, expressed as $O(n^3)$ in big-O notation.

### 2.1 Neural Networks

Neural networks (NNs) are a class of machine learning algorithms that are described as a connected graph of basic compute nodes named neurons. The fundamental compute in each neuron is a dot product of the inputs, named activations, and a set of weights that are unique to that particular neuron. In addition to the dot product, an activation function (tanh, ReLU, sigmoid, etc.) is applied to the output. Equ. 2 shows the operation performed by each neuron.

$$v_j = f(\sum_{i=0}^{n} w_i^j * a_i + b^j) \quad (2)$$

As illustrated in Fig. 2b, a NN is layered such that the output from one layer is passed to each neuron in a subsequent layer, these are named fully connected (FC) layers. The operations for each neuron in a layer can be combined such that the problem is described as a matrix-vector multiplication. For large workloads, multiple inputs

are often batched together and the problem is expressed as a matrix-matrix multiplication. Stacking multiple FC layers together creates a multilayer perceptron (MLP).

Convolutional Neural Networks (CNNs) are a sub-class of NNs designed for image recognition, classification, segmentation and object detection. The main layer in a CNN is the convolution (CONV) layer and is represented as a 3 dimensional block of neurons, often named a filter. Similar to an FC layer, each filter performs a dot product followed by an activation function. The input into a convolution layer is a collection of 2 dimensional input images (or channels) named input feature maps (IFMs). These IFMs are fed into a four dimensional filter array to produce another collection of 2 dimensional output images (or channels) named output feature maps (OFMs). This process is repeated across the image for all filters within the array.

When considering architectures for accelerating deep learning, it is standard practice [2, 4, 19] to use GEMM for recurrent and MLP neural networks given that batching is used across all topologies. However, as presented in a previous study [16], three different methods are commonly used to perform a convolution: (1) GEMM, (2) Winograd and (3) FFT. DeepBench [19] suggests that although convoluation specific functions such as Winograd and FTT exist, GEMM still accounts for over 50% of the highest performing benchmark configurations. Finally, when considering typical data center workloads, previous work [14] has shown that CNNs only account for 5% of the deep learning workloads performed in their data center. Hence by targeting a GEMM for accelerating deep learning we are pursuing an architecture that provides the most benefit for a wide gamut of workloads.

## 2.2 Reduced Precision Networks

With significant research [8, 9, 20–22, 26, 30] indicating that 8 bit or lower precision is sufficient for inference, dedicated hardware such as the Google TPU [14] and the NVIDIA V100 GPU [7], which are optimized for lower precisions have been reported. The benefit of moving to a reduced precision format for neural network computation lies in the efficiency of the multiply and accumulate operations. By moving from single precision floating point to a 32 bit fixed point, normalization is removed and scaling is simplified resulting in smaller hardware. Hence, the area and time complexity is reduced for both the addition and multiplication, improving performance but sacrificing dynamic range. Additionally, by lowering the number of bits, $B$, in the representation, the area requirements of the multiplication and addition generally reduce by factors of $B^2$ and $B$ respectively.

# 3 API

The hardware template presented in Sec. 4 is implemented on the Intel HARPv2 platform with an accompanying software stack and API. As illustrated in Fig. 1, the API contains a high-level function interface for easy integration and a low-level templated interface for fine-grained control. To maintain consistency with other GEMM implementations, the high level API is modeled off other linear algebra libraries. Given Equ. 1 and previous BLAS libraries, the simplified GEMM signature for the FP32 version is:

**Table 1: Tunable Options**

| Parameters | Type | Options |
|---|---|---|
| Systolic Array Size (Sec. 4) | C | *Logic & Memory Limited |
| Precision (Sec. 4.1) | C | FP32, INT16, INT8, INT4, Ternary, Binary |
| Accumulator Width (Sec. 4.1) | C | *Logic & Memory Limited |
| Interleaving (Sec. 4.2) | C&R | *Memory Limited |
| Fused Ops (Sec. 5.1) | C&R | Scaling, Batch Norm, Clip, Rounding, ReLU |

***Limited by the size of the systolic array and available hardware resources. Features are controllable at compile (C) time, runtime(R) or both (C&R)**

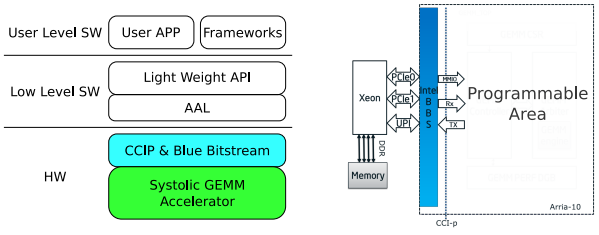
```
void gemm(trans a, trans b, int m, int n, int k, float
    alpha, float* a, int lda, float* b, int ldb, float
    beta, float* c, int ldc);
```

This signature is provided in a set of libraries that is easily compiled into the developers code base. For most projects this should provide sufficient performance as optimizations are performed within the function without interaction from the developer.

Tab. 1 presents a list of the current parameters tunable in the GEMM implementation. Both the precision and accumulator width are configurable at compile time when the systolic GEMM bitstream is generated. If multiple precisions are required for a workload, the API provides a single function for partial reconfiguration that allows for fast precision switching. Post processing fused operations such as value scaling, clipping, rounding and a few deep learning specific operations such as ReLU and Batch Norm can be performed while the results are transfered back to the system memory. These post processing operations are enabled at compile time to be added into the design and can be configured to be bypassed at runtime. For precisions other than FP32, the developer can set the desired accumulator width at compile time, however this does affect memory and logic resource utilization. Similarly, the developer has access to the systolic array interleaving factors. These allow the developer to make fine-grained adjustments to trade off bandwidth with compute efficiency. Sec. 4.2 covers this in more detail. The maximum interleaving level is set at compile time and is bound by the number of memory resources available on the device. However the exact level of interleaving (up to the set maximum) can be controlled at runtime via the lower level APIs.

## 3.1 Runtime Support

The API exposes various configurable parameters to the user-level software. Applications running using our framework leverage the Intel HARPv2 user mode runtime and kernel driver to set these parameters. 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




**(a) Software and Hardware Stack** **(b) Intel HARPv2 Platform**

**Figure 3: (a) The CCIP, blue bitstream and AAL are provided by the HARPv2 platform. (b) The blue bitstream communicates directly with the Xeon and makes memory requests to system memory.**

in Fig. 3a. The hardware template and API chooses between the different memory links, the default setting performs bandwidth balancing between all three. There are three key components in the integration.

```
template<typename T1, typename T2>
void fpga_gemm<T1,T2>::fpga_gemm( trans a, trans b,
int m, int n, int k, float alpha, T1* a, int lda,
T2* b, int ldb, float beta, T1* c, int ldc
int i_a_lead_interleave, int i_b_lead_interleave, int
    i_feeder_interleave, GEMM_MODE i_mode);
```

**Application API:** The low-level functions are templated to support different precisions and modes in Tab. 1. Depending upon the precision, the number of elements packed into cacheline also changes. The low level API is shown above. "a_rows" and "b_cols" refers to the number of rows and columns in A and B matrices respectively. The "common" parameter refers to the common dimension in both the matrices. "i_alpha" and "i_beta" refer to the scaling parameters and "i_mode" refers to the mode in which the hardware accelerator template is set. "i_a_lead_interleave", "i_b_lead_interleave" and "i_feeder_interleave" are the interleaving parameters. These parameters can be used by the runtime to control the memory interleaving and improve the compute efficiency. Internally, the application API uses the AAL user mode runtime to access and initialize the FPGA device. Switching the precision during runtime is supported by the dynamic configuration API.

**Intel AAL:** The AAL layer provides the necessary runtime services and API to access the FPGA device. The framework's API is built on top of the AAL user-mode API's and services. At a very high level, AAL services can be briefly classified into two categories. The AAL user-mode runtime are interfaces that abstract the FPGA hardware via a service oriented model. The AAL kernel-mode driver includes interfaces for allocation of Direct Memory Access (DMA) buffers with shared addressing between the hardware accelerator template and the users 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 AAL kernel mode driver utilizes Intel Blue Bitstream to enumerate the device and perform partial reconfiguration.

**Intel Blue Bitstream (BBS):** Intel BBS is the infrastructure shell component in the FPGA. It abstracts the UPI (Cache Coherent Xeon Link) and the PCIe links to provide a simple load-store like interface called the Cache Coherent Interface (CCI) to the user's accelerator. Intel BBS also handles partial reconfiguration and contains the AAL kernel visible MMIO registers. The AAL kernel driver uses the configuration registers for FPGA device enumeration and initialization.

*3.1.1* ***Heterogeneous Load Balancing.*** The hardware template also supports heterogeneous load balancing. At runtime the workload is partitioned across both the FPGA and CPU. In the case of a GEMM, the A and B matrices are divided into sub blocks and the computation is balanced across the two compute engines. This is useful for particularly large workloads in which the majority of the work is taken by the GEMM function.

# 4 HARDWARE TEMPLATE

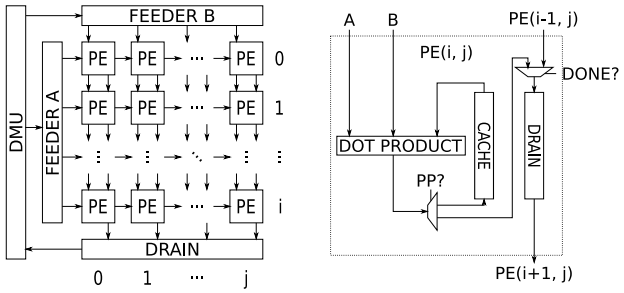
The hardware template illustrated in Fig. 1 contains the systolic GEMM, described below, and several modules handling: memory interleaving, explored in Sec. 4.2, a fused operation scheme and dynamic dot product presented in Sec. 5. As illustrated in Fig. 4a, the hardware template is a systolic array of processing elements (PEs), each containing a dot product module and two memory buffers, named the cache buffer and drain buffer. The systolic array operates by iteratively processing chunks of the input matrices stored in the feeders. There are two orthogonal feeders that connect to their respective edges of the array. The design is fully pipelined, hence each cycle data is fed into the array through the feeders and propagated along the appropriate rows and columns. The feeders are by default double buffered to ensure that multiple read requests are in-flight to saturate the system bandwidth and that compute stalls due to insufficient memory are minimized. The data management unit (DMU) is responsible for requesting the input data, filling the feeders, draining out completed sections of the compute and generating write requests to the system memory. Within the systolic array, input vectors are interleaved into each PE to take advantage of data reuse and help meet the bandwidth requirements of the system. Since the input is interleaved, a small cache within each PE is necessary to store the partial results for accumulation later in the computation.

The feeders are memory modules that manage the flow of data into the array. Each feeder is by default double buffered such that one buffer can be operated while the other is being filled. The number of buffers is configurable at compile time and can be used to alleviate issues related to periods of inconsistent bandwidth.

The drain is a large interconnect that controls the flow of data down the columns of the systolic array. When the signal to drain is given, the 'drain' memories at the bottom of the array start to empty. Each column acts as a large FIFO that produces a result every cycle.

## 4.1 Processing Element

The PE contains a dot product module with two memories, a partial results 'cache' and completed results 'drain'. Input vectors are passed into each PE every cycle where the dot product is performed and any partial results are accumulated. In cases where the dot




(a) Systolic Array (b) Processing Element

**Figure 4: The hardware template: (a) The array size is configurable with one limitation, the drain bus width must be $\leq$ 64 bytes. For FP32 this limits the number of columns to $j = 16$. (b) This is a PE for a given row ($i$) and column ($j$).**

product is larger than the input vector length, the partial result is stored in the cache which will be used later in the computation. If the dot product length is smaller than the input vector length, or more commonly the final partial input vectors have been passed in to the PE, the completed result is stored in the drain and is ready to be taken out of the array. To ensure high throughput, reading out the array, i.e. 'draining', can be performed while partial results of the next chunk are produced and stored in the cache, as both memories operate independently. The one exception is when a set of complete results would be written into a non-empty drain. In this case the computation is stalled until the drain is empty. The array control signals for the computation stage are passed across rows whereas the control for the draining is passed across columns. Each PE is responsible for passing data in a linear fashion along its row and column. Additionally, each PE is fully pipelined such that the result of a single dot product is produced every cycle.

Depending on the desired bit width, the dot product is either performed in DSPs, constructed using logic resources or a combination of both. The PE currently supports single precision floating point (FP32), 16, 8, 4 and 2 bit Integer and Fixed Point (INT16, INT8, INT4, INT2) and more exotic data types for deep learning workloads: INT16xTernary, INT8xTernary, BinaryxBinary (BINxBIN).

For the integer and fixed point data types, the bit width of the each stage of the adder tree in the dot product is increased by one. Apart from FP32, the accumulator bit width is configurable at compile time for all data types. After accumulation and before storing into the 'cache' or 'drain', the results are truncated and rounded. Currently stochastic rounding and round to nearest are supported by the framework and are configurable at compile time.

## 4.2 Blocking and Interleaving

During GEMM each element in matrix A is used $n$ times and each element in matrix B is used $m$ times. The bandwidth can be minimized by storing both A and B on-chip and reusing each element. However, the number of on chip memories quickly becomes the limiting factor when dealing with larger matrices. To handle larger matrix sizes both A and B are partitioned into chunks and sent down in batches. This is usually referred to as blocking and is a standard practice in GEMM implementations to achieve high performance.

Interleaving on the other hand, is an architecture specific optimization designed to enable data reuse on a fine-grained level. It takes advantage of the data reuse in a GEMM and operates by feeding the same vectors into the PEs in a specific order. Both the leading dimension of matrices A and B, $m$ and $n$ respectively, have independent interleaving factors that are controlled at both compile and run time. Fig. 5 shows a simplified example, 1x1 array, of how interleaving operates within our hardware template. The interleaving factors for feeder A and B are 3 and 2 respectively. Each row in Feeder A and column in Feeder B are partitioned into two separate blocks, $a_x$ and $b_x$ respectively, the result of these partitions are accumulated to create the final 3x2 matrix.

At $t = 0$, $a_0$ and $b_0$ are passed into the PE and the partial result is stored in the first location in the cache. At $t = 1$, the pointer to memory location $a$ is incremented, $b_0$ is reused and $a_1$ is passed into the PE; with the partial result stored in the second location in the cache. This continues on to $t = 3$ where the $a$ memory pointer

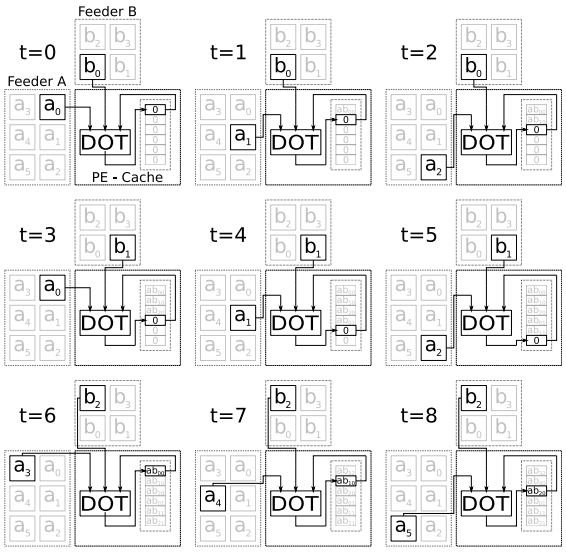



**Figure 5: Interleaving Example: This shows a simplified example of how the PE operates and the concept of interleaving.**

is reset back to zero and the $b$ memory pointer is updated. Now $a_0$ is reused and $b_1$ is passed into the PE; the partial result is stored in the fourth location in the cache. At $t = 6$, the first column and row of A and B respectively, have been processed and the pointers move on to the second row and column. Instead of accumulating zeros, the previous value from $t = 0$, $a_0 b_0$, is added to the result of $a_3$ and $b_2$ and is stored back into the first location. This process continues in a similar fashion for $t = 7, ..., 11$ until all rows and columns have been processed.

Both blocking and interleaving are used together to improve performance. The blocking size is determined by the interleaving factor as well as the number of row and columns in the systolic array. Initially the interleaving size was fixed at compile time. While the systolic array would perform at near peak theoretical performance for large matrices, for smaller matrix sizes there was a significant decrease in performance. Since both matrix A and B would need to be padded with zeros until they were a multiple of the block size. We observed that by adding configurable memory interleaving support at runtime, the performance significantly improved as discussed later in Sec. 6.1.2. In most cases the optimal interleaving size can be directly calculated using $I = \min_{x_L,...,x_H} modulo(DIM, x)$. $x$ is the range of different interleaving values supported by the hardware template, and $DIM$ is the leading dimension of either the $A$ or $B$ input matrices. Hence the size of the block's leading dimension for A and B can be calculated using $S_A = I_A * HW_{ROWS}$ and $S_B = I_B * HW_{COLS}$ respectively.

Now, given the dot product size ($S_D$) and the length of the block's common dimension ($S_C$), the number of cycles per block can be calculated using the A and B interleaving sizes ($I_A$ and $I_B$) as illustrated in $Cycles = I_A * I_B * \frac{S_C}{S_D}$. Given the block sizes of A ($S_A$x$S_C$), B ($S_B$x$S_C$) and C ($S_A$x$S_B$) and $Cycles = I_A * I_B * \frac{S_C}{S_D}$, the read and write bandwidth requirements can be calculated directly using Equ. 3

$$Bandwidth = f\frac{Bytes(S_A S_C + S_B S_C + S_A S_B)}{Cycles} \quad (3)$$

where $f$ is the operating frequency of the design and *Bytes* is the number of bytes used to represent each element of A, B and C.

## 5 DEEP LEARNING

This section discusses the deep learning specific optimization and hardware template features available in our framework. We describe the fused operations and dynamic dot product modules from Fig. 1.

### 5.1 Fused Operations

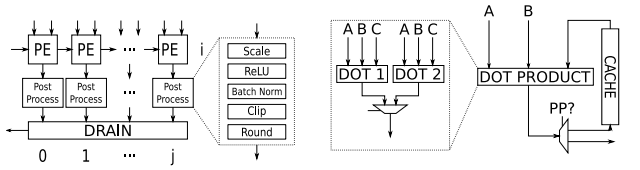
In the context of deep learning, often additional operations such as the activation function or batch normalization are performed after the FC or CONV layers.

As presented in Fig. 6a, as the results are drained out of the systolic array a post processing module can apply some basic operations. The post processing module contains a common interface such that extra functions such as sigmoid, tanh or custom scaling schemes can be added without modification to the systolic array.

### 5.2 Dynamic Dot Product

Training neural networks on FPGA hardware has been challenging since the gradient update step requires a higher precision. Recent work [17] has shown that through the use of a novel quantization scheme, hardware friendly backprogation is possible if a mixed precision FP32x(Binary/Ternary/Integer) dot product can be performed. To support this we added the ability to dynamically switch between dot product types during runtime. Fig. 6b presents the necessary change to the PE. To illustrate the advantages of this change, we implemented a BINxBIN dot product and a FP32xFP32 dot product. With the addition of dynamic dot product switching, both training and inference is supported on the FPGA.

For a typical layer in a state-of-the-art BNN [17] the FPGA performs the BIN X BIN operations very efficiently, hence the CPU can be freed during that time to perform other tasks in its pipeline. Tab. 2 illustrates the required operations of the middle and end portions for a binarized AlexNet. It shows that different layers can be partitioned, for both forward and backward passes, across the FPGA, CPU or using heterogeneous load balancing (FPGA+CPU). The RELU operation during the forward and backward pass can be performed on the FPGA or CPU depending on which device the result was calculated on. For the batch norm operation the forward pass can be performed on the FPGA as it is a scale and shift operation. At this time the hardware template does not support the backward computation of batch norm or pool operations and adding support is future work. To further support mixed precision,




(a) Fused Operations

(b) Dynamic Dot Product Switching

**Figure 6: (a) The post processing module is configurable at compile time and runtime. (b) Dynamic dot product switching allows for greater micro-architecture exploration and flexibility when designing for reduced precision networks.**

**Table 2: Mixed Precision Inference and Training**

| Layer | Location | | Type | |
|---|---|---|---|---|
| | Forward | Backward | Forward | Backward |
| ... | ... | ... | ... | ... |
| conv | FPGA | FPGA+CPU | BINxBIN | FPxBIN |
| c&r | FPGA | N/A | INT | STE |
| relu | FPGA | FPGA+CPU | INT | FP |
| norm | FPGA | CPU | FP | FP |
| pool | CPU | CPU | FP | FP |
| ... | ... | ... | ... | ... |
| fc | CPU | CPU | FPxFP | FPxFP |
| prob | CPU | CPU | FP | FP |

**For the standard configuration of a BNN, a mixed precision implementation on the Xeon+FPGA utilizing the dynamic dot product could operate in this manner. The straight through estimator (STE) is used for the clipping and rounding (c&r) layer, hence no operation is required on the backwards pass.**

in the future we plan to include a hybrid 2-D systolic array with mixed precision PEs between different rows or columns to alleviate routing congestion due to multiplexing between DOT types.

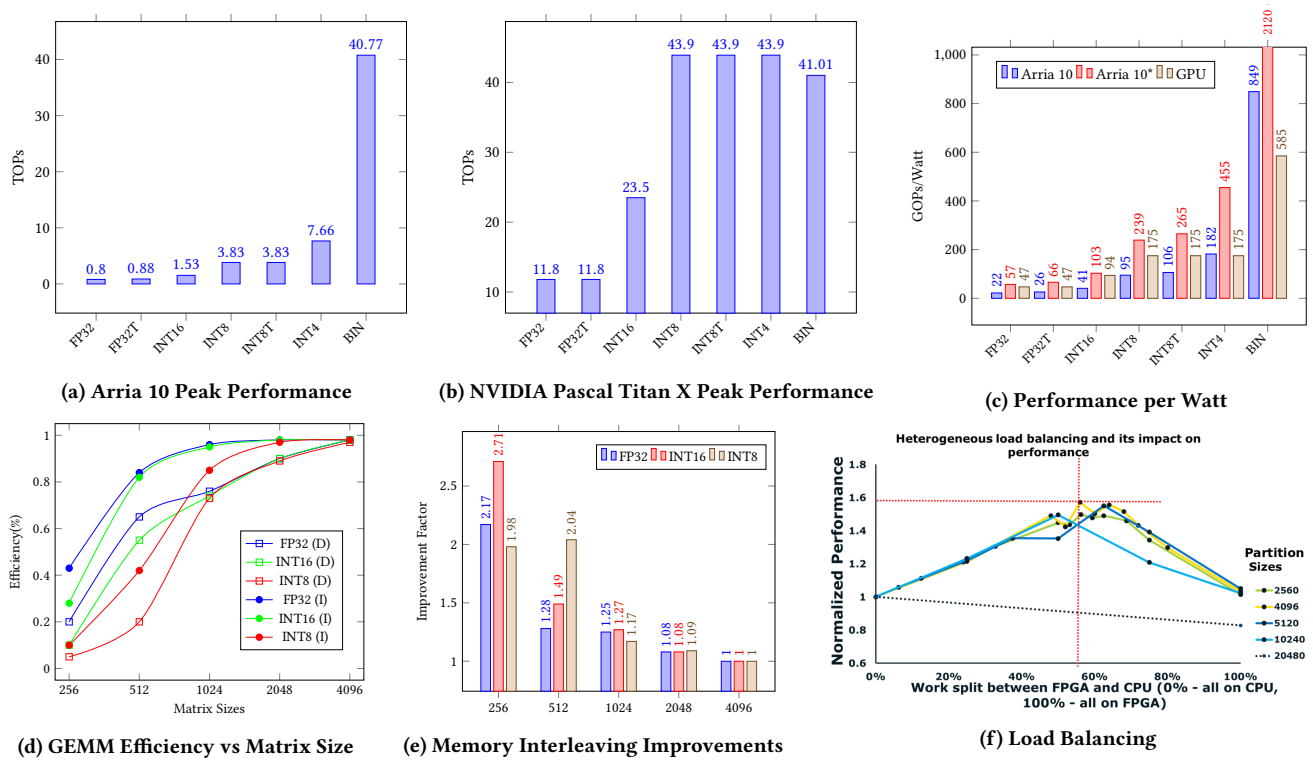
## 6 RESULTS

The results are presented in two section, first we show the performance of the GEMM for the various precisions and compare to a NVIDIA Titan X Pascal GP102 GPU on 16nm process. We illustrate the benefit of memory interleaving on various matrix sizes for three precisions, FP32, INT16 and INT8. We present our results of heterogeneous load balancing and the effect of the different partitioning sizes across the FPGA and CPU. The second section focuses on the deep learning workload. Results for three different binary network topologies, (AlexNet [15], VGGnet [25] and ResNet [13]) are presented and are evaluated against the GPU. We illustrate the impact of memory interleaving on AlexNet at various batch sizes for FP32 and discuss further possible optimizations. Finally, we investigate several possible mixed precision implementations and evaluate their performance for both training and inference. All FPGA results were gathered by measuring the wall time of the function call for each configuration on the HARPv2 system.

Since the systolic array size is configurable at compile time, it can be tailored to the resources available on the FPGA. In the case of the HARPv2, all of the available area was used to implement the hardware template minus the area taken by the Intel BBS. However, other modules could be implemented along side the hardware template at the cost of array size. For FP32, this results in 160 PEs, with an array size of 10 rows and 16 columns, operating at 312.5Mhz. With the dot product size configured to 8, this uses 1280 of the 1285 DSP available. Even though our measurements are performed on the HARPv2, the hardware template, written in SystemVerilog, is only dependent on the CCI interface and can be implemented on larger or smaller FPGAs.

### 6.1 GEMM

Fig. 7a presents the performance of the GEMM architecture presented in Sec. 4 on the Arria 10. Since a floating point addition and multiplication is needed for all FP32 modes, all operations are




**(a) Arria 10 Peak Performance** **(b) NVIDIA Pascal Titan X Peak Performance** **(c) Performance per Watt**

**(d) GEMM Efficiency vs Matrix Size** **(e) Memory Interleaving Improvements** **(f) Load Balancing**

**Figure 7: GEMM MCP Performance. (a) shows the current peak performance of the framework for a few selected precisions. (b) gives the GPU results for same precision selected in (a). (c) shows the current performance per watt for both the GPU and Arria 10 as well as *modified Arria 10 results which normalize for process node. (d) & (e) illustrates the difference between the framework with (I) and without (D) memory interleaving. (f) illustrates the load balancing results.**

performed in the DSP. While it is possible to implement these operations in logic it quickly becomes very expensive and the design is constrained by routing resource when increasing the array size. Hence the performance is limited by the number of DSPs available on the chip. For FP32T (FP32xTernary) further optimizations can be made by removing the multiplication and implementing a simplified multiplexer unit.

For integer precisions we observe a better than linear scaling of performance. As the bit width becomes smaller, the dot product is a good fit for the FPGA architecture. INT16 doubles the FP32 performance since each DSP can perform two multiplications and two additions. The hardware template supports the use of logic resources when implementing a larger array. However specifically for INT16, the multiplication utilization and routing resources become a significant issue and only result in a small improvement in the peak tera-operations per second (TOPs). Since the DSP architecture does not natively support 8 bit operations, doubling the performance is achieved by using a dot product built out of both DSP and logic elements. For INT8 and INT4, extra rows are added to the array since there is sufficient logic resources available to implement additional math operations. Moving to INT8T (INT8xTernary) provides a performance per watt improvement over INT8 since: (1) each multiplication is replaced by a multiplexer (2) with the removal of the multiplication, the accumulator bit width can be reduced. The BIN GEMM is implemented completely in logic and uses an XOR and lookup-table based dot product.

*6.1.1 GPU Comparison.* GPUs are known for their linear algebra performance as shown in Fig. 7b, reported performance from previous studies [17, 21]. Compared to the Arria 10, the GPU has higher raw performance for all cases apart from the binary GEMM. Considering power, the FPGA shows superior performance for binary and 4 bit integer which are not a good fit for the GPU architecture. However considering that the GPU is at a newer processes node than the FPGA, TSMC 16nm and 20nm respectively, we have plotted a normalized Arria 10 result of the same design and frequency while cutting power by 60% [6]. In the normalized case, the FPGA out performs the GPU especially for the binary GEMM.

*6.1.2 Memory Interleaving.* Efficiency is calculated by comparing the measured TOPs to the theoretical maximum value for a given design. The theoretical TOPs are calculated by simply taking the number of compute units and multiplying by the frequency, disregarding any time for data transfer. Fig. 7d and Fig. 7e illustrate the efficiency of the array at different matrix sizes as well as the improvement of the memory interleaving optimization. The sizes tested were square matrices of the x axis labels, i.e., for 256 the A, B and C matrices are all 256x256.

For the smallest matrix size 256, we can see that for the unoptimized designs (D) the efficiency for FP32, INT16 and INT8 are all below 20%. In these cases the low efficiency is caused by an inefficient use of blocking. Memory interleaving alleviates these issues as per Equ. 3. With runtime configurable memory interleaving (I), presented in Sec. 4.2, we see a 2.7x improvement in efficiency for

the smaller matrix sizes. In some cases even with memory interleaving, the efficiency of the design is quite low. Usually the data transfer of the next chunk in the input matrix is hidden during the computation of previous chunks. For smaller matrix sizes, the number of chunks are small and hence the transfer cost cannot be amortized by the compute. Additionally, the initial transfer time for the first chunks of A and B as well as the transfer of C account for the majority of the measured execution time. This issue can be resolved by staging multiple GEMMs such that transfer of the A, B and C chunks overlap with the compute from the previous GEMM. This is similar to how larger single GEMMs are performed. For matrix sizes past 512, the efficiency for all precision is above 80% and becomes 90% past 1024 where it quickly reaches peak performance. For square matrix size 4096 the effectiveness of memory interleaving has been diminished, however as discussed later in Sec. 6.2.2, non-square matrix sizes still see significant improvements.

*6.1.3 Heterogeneous Load Balancing.* Fig. 7f presents the performance of the FP32 GEMM when load balancing is performed over the FPGA and a 14 core Xeon CPU. Peak performance is achieved as the work split approaches 60% on the FPGA and 40% on the CPU. In this configuration the FPGA and CPU contribute similar peak performance. This is consistent for most partitioning sizes apart from 20480. In this case, the whole compute is performed on either the CPU or the FPGA. For small partitioning size either the CPU or FPGA becomes memory bound and peak performance drops. Interestingly, for this particular workload the optimal partitioning size is 4096, showing that a finer-grained partitioning of the problem performs better than coarse-grained sizes such as 10240. This illustrates that the CPU and FPGA working in tandem can achieve a 1.6x improvement in performance over a 14 core implementation.

## 6.2 Neural Network Evaluation

In this section we extend on previous work [18] and apply our customizable GEMM framework to three deep learning workloads: AlexNet [15], VGGnet [25] and ResNet [13]. While the GEMM targets many different precisions, we specifically target binary neural networks since: (1) from Fig. 7a, this clearly offers the best performance over a GPU and (2) recent work [17] has shown that implementations can achieve high accuracy even for binary weight and activation networks. Additionally, we provide a study on the effectiveness of memory interleaving on layer efficiency for the AlexNet topology running in FP32. Finally, we present a mixed precision training and inference scheme targeted at leverage heterogeneity and the dynamic dot product module

The evaluation was performed on both the CONV and FC layers for inference with the standard mini-batch size used for each topology. The total network performance and Images per Second (IPS) is calculated based on the weighted average of the layers operation contribution to the overall network. A runtime breakdown of each topology was collected using the 14 core Xeon Broadwell CPU running Caffe with Intel MKL2017.

**Table 3: Network Performance**

| Device | FPGA | | | | GPU | | | |
|---|---|---|---|---|---|---|---|---|
| | TOPs | GOPS/W | IPS | IPS/W | TOPS | GOPs/W | IPS | IPS/W |
| AlexNet | 31.54 | 657.27 | 1610 | 33.54 | 37.60 | 568.09 | 1626 | 25.02 |
| VGGNet | 31.18 | 649.67 | 114 | 2.39 | 35.85 | 522.59 | 121 | 1.78 |

*6.2.1 Binary Network Performance.* As shown in Fig. 7a, Fig. 7b and Fig. 7c the FPGA and GPU achieve 40.77 TOPs and 41.01 TOPs for the binary GEMM respectively, which is within 99% of their theoretical peak performance. Compared to the GPU, the FPGA achieved 849.38 GOPs per Watt which results in a 1.44x improvement in energy efficiency over the GPU at 585.86 GOPs per Watt.

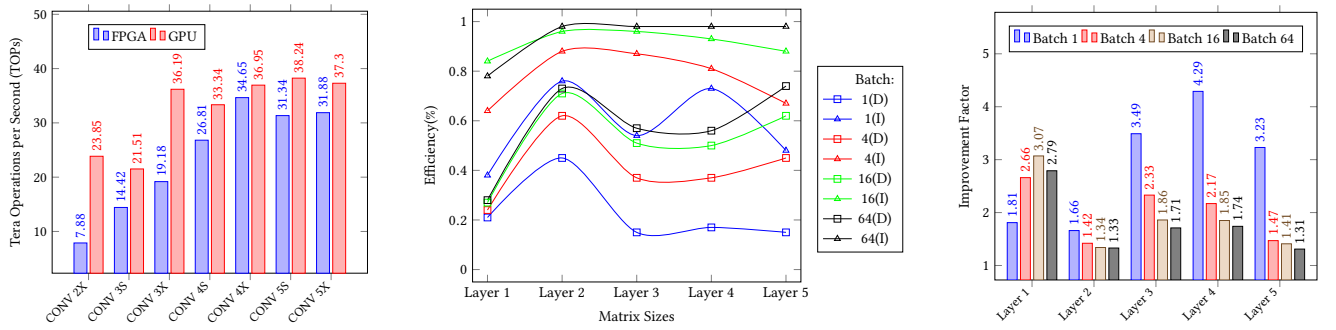
As illustrated in Tab. 3, the FPGA, without load balancing, achieves 83% and 86% of the GPU raw performance for AlexNet and VGGNet. However in terms of energy efficiency (GOPs/Watt) the FPGA surpasses the GPU by 1.15x and 1.23x for AlexNet and VGGnet respectively. When extrapolating to Images per Second (IPS) the difference in network performance is less stark with the FPGA achieving 99% and 94% of the GPUs IPS for AlexNet and VGGNet respectively. Hence, the FPGA shows better Images per Second per Watt compared to the GPU for both AlexNet and VGGNet resulting in a 1.34x improvement for both topologies.

For ResNet-34 the FPGA achieves 70% of the GPU performance at 23.47 and 33.34 TOPs respectively. However it is on par for energy efficiency at 489.05 GOPs/Watt for the FPGA and 485.68 GOPs/Watt for the GPU, showing a 1.03x improvement. This drop in FPGA performance compared to AlexNet can be understood by examining the layer breakdown presented in Fig. 8a. The GPU out performs the FPGA for the first three layer sets which contribute 50% of the total operations. While increasing the batch size can alleviate some of this discrepancy, this is undesirable since it can affect training rate and total execution time.

If we examine specifically the first layer, the FPGA achieves 7.88 TOPs, which is 19% of the measured peak performance presented in Fig. 7a. The main cause of this inefficiency is introduced by the padded zeros in the common dimension of the input matrices. As future work we are planning to enable runtime configurable memory interleaving for the common dimensions of the matrices. This should result in significant performance improvements since the majority of the padded zero computations will be avoided.

*6.2.2 Effect of Memory Interleaving on AlexNet.* Fig. 8b and Fig. 8c show the performance of the first five convolutional layers in AlexNet for FP32. Different batch sizes ranging from 1 to 64 illustrate the performance improvement that is provided by memory interleaving. For a batch size of 1, A and B are long-skinny/short-fat matrices. Hence with fine control of the interleaving we can see 1.8x and 1.6x improvement for the first two layers, while for layer 3,4 and 5 the improvement is over 3x and up to 4x. For batch sizes of 4, 16 and 64, the A and B matrices are becoming increasingly square. Hence we don't see the same level of improvement, however it achieves near peak performance. The improvement for the first layer is more significant than the other since the first layer exhibits the worst long-skinny/short-fat matrix sizes. However it is clear that memory interleaving significantly improves the efficiency, achieving a 1.3x up to 4x improvement.

*6.2.3 Dynamic Dot Product.* As discussed in Sec. 5.2, mixed precision GEMMs are needed to handle both the forward and backwards pass. Typically, the first convolution and last fully connected layers are performed at full precision while the inner layers are




**(a) Binary ResNet Layer Performance** **(b) FP32 AlexNet Efficiency** **(c) FP32 AlexNet Efficiency Improvement**

**Figure 8: (a) Binarized ResNet layer performance. (b) & (c) Breakdown of static design (D) vs configurable interleaving (I) for different batch sizes on AlexNet at FP32 precision.**

performed at lower precision [17]. As shown in Tab. 2, during the backward pass, the gradients are computed using single precision floating point hence the operation is a FP32xBIN GEMM. While our framework contains an API for switching between different precisions (Sec. 3.1) the latency is determined by the partial reconfiguration (PR) time. In the cases where the PR latency is too high, a dynamic dot product may be more appropriate.

By examining the breakdown for binarized AlexNet, similar to Tab. 2, we designed three FPGA implementations, two with dynamic dot product modules, targeting different portions of the topology: (1) a single BINxBIN, (2) a BINxBIN and FPxFP, (2H) a version of (2) that performs the FPxFP GEMM utilizing heterogeneous load balancing (FPGA+CPU) and finally (3) a BINxBIN and FPxBIN. All other layer operations not supported are assumed to be implemented in software on all cores of the Xeon. Implementation (1) and (2) were synthesized for the HARPv2 whereas (3) was extrapolated using the results of (2) and the analysis from Sec. 6.1 (2) is able to perform the FPxBIN operations by representing the BIN values as floating point. Inference and training take 421*ms* and 892*ms* respectively, implementation (1) accelerates the BINxBIN operations which account for 272*ms* of the execution time (all in the forward pass). This corresponds to 65% and 30% of the total interference and training time respectively. (2) accelerates the most layers with the BINxBIN and FPxFP operations accounting for 327*ms* (77%) and 735*ms* (82%) of the total execution time. Finally, (3) accelerates the BINxBIN and FPxBIN operations accounting for 272*ms* (65%) and 593*ms* (66%) of inference and training time respectively.

As illustrated in Tab. 4, for pure inference, implementation (1) achieves the best results, reporting the fastest forward execution time. However for training (Forward+Backward), implementation (2H) and (3) show a greater speedup, achieving 1.67x and 1.41x improvement over a software only implementation. Implementations (2), (3) and (2H) use the dynamic dot product, hence the BINxBIN performance suffers since a smaller systolic array size is used to accommodate the extra routing resources. Although implementation (2) implements all operations on the FPGA, this reduces the amount of hardware that can be dedicated to the bottlenecks, resulting in the lowest speed up, 1.23x. The CPU+FPGA implementations (1), (3) and (2H) show that the best overall performance is achieved by leveraging the CPU and specializing the implementations for only the most profitable parts of the algorithm, or by utilizing heterogeneous load balancing. Compared to other CPU+FPGA platforms, the same network performance would not be achieved since SoC type CPUs have relatively low FLOPS compared to Xeon CPU.

**Table 4: Implementation Peak Performance**

| Impl. | BINxBIN (TOPs) | FPxFP (TFLOPs) | FPxBIN (TFLOPs) | Forward (ms) | Backward (ms) | Total (ms) |
|---|---|---|---|---|---|---|
| SW | - | - | - | 421 | 471 | 892 (1x) |
| (1) | 40.77 | 0 | 0 | 158 (F) | 471 (C) | 630 (1.41x) |
| (2) | 25.4 | 0.8 | 0.8 | 164 (F) | 537 (F) | 702 (1.23x) |
| (3) | 25.4 | 0 | 0.88 | 165 (F) | 502 (F) | 668 (1.33x) |
| (2H) | 25.4 | 1.4 | 1.4 | 163 (F+C) | 369 (F+C) | 533 (1.67x) |

**Where possible, the operation was performed on the FPGA (F), the 14 core CPU (C) or using heterogeneous load balancing (F+C).**

## 7 RELATED WORK

Compared to other work, our hardware template in the BINxBIN configuration achieves the highest peak TOPs, MOPs / LE and the second highest GOPs / Watt. We appreciate that comparing across different devices and manufacturers is difficult and try to ensure that accurate comparisons are made. While the results reported in [26] are for an older process node and lower frequency, [10] shows their framework performance on a device of similar size and process node as our own. It is expected that [26] should achieve better energy efficiency since the device is targeted at low power SOC applications, while ours is a data center target device. One key advantage of our hardware template is that it provides several other data types and customizations that benefit applications outside of deep learning. Additionally, with the Xeon CPU able to provide high floating point compute power, any changes to the underlying neural network algorithms, such as those presented in [17] can be supported without changes to the hardware architecture.

### 7.1 Existing accelerators on Xeon+FPGA

As the Xeon+FPGA platform continues to gain popularity there has been prior work that studied acceleration on this platform, such as [28, 29]. [29] studied CNN with math optimization (e.g., FFT transformation). [28] studied irregular pointer chasing applications. Heterogeneous CPU-Accelerator platforms are quickly becoming pervasive throughout computational systems and clusters. With the fast adoption of machine learning and deep learning in business, the

**Table 5: Previous Work**

| | [26] | [10] | [30] | Our Work |
|---|---|---|---|---|
| Platform | Zynq z7045 | Kintex US KU115 | Zynq 7Z020 | Arria 10 GX1150 |
| Logic Elements (LEs) | 350K | 1,451K | 85K | 1,150K |
| Power (W) | 11.3 | 41 | 4.7 | 48 |
| TOPs (Peak) | 11.612 | 14.8 | 0.32 | 40.77 |
| MOPs / LE | 33.17 | 10.19 | 4.43 | 35.45 |
| GOPs / Watt | 1027.68 | 360.97 | 44.2 | 849.38 |

computation requirements of cloud and local distributed systems is increasing at an exponential rate.

Several studies [5, 24, 27] have focused on key workloads to better understand the requirements of these algorithms and their performance on CPU+FPGA systems. [5] provides a quantitative analysis of a QPI based CPU+FGPA system compared to a PCI-E based CPU+FPGA system. Key differences between the two platforms such as different memory models and peak bandwidth were highlighted and a decision tree based flowchart was provided as a guide to assist developers when choosing a platform.

## 7.2 Existing NN FPGA accelerators

Previous work has been done on NN accelerators [3, 11, 20, 21, 26, 30] and our work differs in these key areas. We studied a flexible hardware template where the CPU and FPGA work in tandem to accelerate only the functions well suited to each device. Previous work has mainly focused on inference while we examined potential mixed precision implementations that accelerate both inference and training. Training is more complex than inference, as it requires more variety of operations. We showed that an "All-FPGA approach" may yield sub-optimal performance when compared to a tailored or heterogeneous approach. We handled training of NNs by relying on the Xeon CPU to leverage the existing x86 software ecosystems.

# 8 CONCLUSION

We presented a customizable matrix multiplication framework that includes a simple software API and a hardware template for designing custom GEMM accelerators on the HARPv2. We demonstrated that for deep learning workloads, the FPGA was either on par or exceeded what the GPU could perform in the case of binary neural networks and offered insight into some of the issues faced when designing for this system. We evaluated several heterogeneous implementations and illustrated that by dedicating FPGA resources to accelerating specific bottlenecks and utilizing the CPU for other operations, results in higher performance implementations. While the HARPv2 is still an emerging technology, it is able to stay competitive with a high performance discrete GPU by taking advantage of close collaboration between the FPGA and CPU.

# ACKNOWLEDGMENT

This research was supported under the Australian Research Councils Linkage Projects funding scheme (project number LP130101034) and Zomojo Pty Ltd.

# REFERENCES

[1] 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. https://doi.org/10.1145/567806.567807
[2] Firas Abuzaid, Stefan Hadjis, Ce Zhang, and Christopher Ré. 2015. Caffe con Troll: Shallow Ideas to Speed Up Deep Learning. *CoRR* abs/1504.04343 (2015). http://arxiv.org/abs/1504.04343
[3] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. 2017. An OpenCL (TM) Deep Learning Accelerator on Arria 10. In *ISFPGA*.
[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). http://arxiv.org/abs/1410.0759
[5] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *DAC*.
[6] Taiwan Semiconductor Manufacturing Company. 2013. TSMC 16/12nm Technology. (2013). http://www.tsmc.com/english/dedicatedFoundry/technology/16nm.htm
[7] NVIDIA Corporation. 2017. *NVIDIA TESLA V100 GPU ARCHITECTURE.* Technical Report WP-08608-001. NVIDIA Corporation. https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf
[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. In *NIPS*.
[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830* (2016).
[10] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Scaling Binarized Neural Networks on Reconfigurable Logic. In *PARMA-DITAM*.
[11] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *FCCM*.
[12] PK Gupta. 2016. Accelerating Datacenter Workloads. In *FPL*.
[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.
[14] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, and et. al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *CoRR* abs/1704.04760 (2017). http://arxiv.org/abs/1704.04760
[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*.
[16] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). http://arxiv.org/abs/1509.09308
[17] Asit K. Mishra, Eriko Nurvitadhi, Jeffrey J. Cook, and Debbie Marr. 2017. WRPN: Training and Inference using Wide Reduced-Precision Networks. *CoRR* abs/1709.01134 (2017). http://arxiv.org/abs/1709.01134
[18] Duncan Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Suchit Subhaschandra, and Debbie Marr. 2017. High Performance Binary Neural Networks on the Xeon+FPGA Platform. In *FPL*.
[19] Sharan Narang. 2016. DeepBench. (2016). https://svail.github.io/DeepBench/
[20] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. 2016. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *FPT*.
[21] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *ISFPGA*.
[22] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: Imagenet Classification Using Binary Convolutional Neural Networks. In *ECCV*.
[23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y
[24] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*.
[25] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv:1409.1556* (2014).
[26] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *ISFGPA*.
[27] Xuechao Wei, Yun Liang, Tao Wang, Songwu Lu, and Jason Cong. 2017. Throughput optimization for streaming applications on CPU-FPGA heterogeneous systems. In *ASP-DAC*. IEEE.
[28] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C Hoe. 2016. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *ISFPGA*.
[29] Chi Zhang and Viktor Prasanna. 2017. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *FPGA*.
[30] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *ISFPGA*.