Sanaullah *BMC Bioinformatics* 2018, **19**(Suppl 18):490
https://doi.org/10.1186/s12859-018-2505-7

BMC Bioinformatics

RESEARCH Open Access

Real-time data analysis for medical diagnosis using FPGA-accelerated neural networks

Ahmed Sanaullah[1], Chen Yang[1], Yuri Alexeev[2], Kazutomo Yoshii[3] and Martin C. Herbordt[1*]

From Computational Approaches for Cancer at SC17
Denver, CO, USA. 17 November 2017

Abstract

Background: Real-time analysis of patient data during medical procedures can provide vital diagnostic feedback that significantly improves chances of success. With sensors becoming increasingly fast, frameworks such as Deep Neural Networks are required to perform calculations within the strict timing constraints for real-time operation. However, traditional computing platforms responsible for running these algorithms incur a large overhead due to communication protocols, memory accesses, and static (often generic) architectures. In this work, we implement a low-latency Multi-Layer Perceptron (MLP) processor using Field Programmable Gate Arrays (FPGAs). Unlike CPUs and Graphics Processing Units (GPUs), our FPGA-based design can directly interface sensors, storage devices, display devices and even actuators, thus reducing the delays of data movement between ports and compute pipelines. Moreover, the compute pipelines themselves are tailored specifically to the application, improving resource utilization and reducing idle cycles. We demonstrate the effectiveness of our approach using mass-spectrometry data sets for real-time cancer detection.

Results: We demonstrate that correct parameter sizing, based on the application, can reduce latency by 20% on average. Furthermore, we show that in an application with tightly coupled data-path and latency constraints, having a large amount of computing resources can actually reduce performance. Using mass-spectrometry benchmarks, we show that our proposed FPGA design outperforms both CPU and GPU implementations, with an average speedup of 144x and 21x, respectively.

Conclusion: In our work, we demonstrate the importance of application-specific optimizations in order to minimize latency and maximize resource utilization for MLP inference. By directly interfacing and processing sensor data with ultra-low latency, FPGAs can perform real-time analysis during procedures and provide diagnostic feedback that can be critical to achieving higher percentages of successful patient outcomes.

Keywords: FPGA, Machine learning, Multi-layer perceptrons, Real-time, Inference, Cancer, Mass-spectrometry

*Correspondence: herbordt@bu.edu
[1]Computer Architecture and Automated Design Lab, Boston University, Boston, MA, USA
Full list of author information is available at the end of the article

Background

Machine learning (ML) plays an integral part in solving many key scientific problems. It has been applied to such varied domains as finding cancer treatments [1], weather simulations [2], and design of new nanocatalysts [3]. Machine learning methods have been used in medical applications for many years [4]. First of all, machine learning can be used in monitoring patients' vital signs. Muller, et al. introduced a Brain-Computer Interface (BCI) [5] based technique that can extract appropriate features from continuous EEG signal [6]. The proposed system collects a number of trials from patients while they are asked to perform fixed tasks. Using the collected datasets as training data, the system infers the typical EEG patterns in real-time. In [7], Shoeb and Guttag presented a machine learning algorithm to recognize epileptic seizure from the scaled EEG signal. Machine learning is also widely used in medical image retrieval, enhancement, processing and mapping as introduced in [8–12]. Zacharaki, et al. performed a study on distinguishing different types of brain tumors based on Support Vector Machine Recursive Feature Elimination (SVM-RFE) algorithm [13]. Pereira et al. introduced an approach to use machine learning algorithms to decode variables of interest from fMRI data [14]. Apart from that, machine learning is also an effective supplement in diagnosing diseases [15, 16]. Ozcift, et al. constructed Rotation Forest (RF) ensemble classifiers for multiple machine learning algorithms in a computer-aided diagnosis (CADx) system that is used for diagnosing diseases like diabetes and Parkinson [17]. Li and Zhou proposed a semi-supervised learning algorithm, Co-Forest, that uses undiagnosed samples along with only a small amount of diagnosed ones as training datasets for CAD systems targeting breast cancer diagnosis [18].

Multi-Layer Perceptrons (MLPs) are an important subset of ML that not only optimize existing applications and practices but also widen that pool by enabling designs to meet more stringent requirements of reliability, performance, complexity, and portability that are not possible from traditional approaches. MLPs consist of multiple layers of firing neurons, with each layer using responses of previous neurons as stimulus. Use of MLPs is divided into two stages: training and inference. Training is an iterative process that determines neuron connection strengths in a given MLP. We assign initial values to connection strengths and then apply training cases as inputs to the model. The correct results of training cases are known beforehand, which we refer to as expected values. Corresponding measured outputs of the model are compared with these expected values, and connection strengths are then updated by a (static or dynamic) factor in order to minimize the error between the two sets of results. This process is repeated until measured results converge to values that reduce errors to within acceptable bounds. Training is typically an offline operation and thus does not impact analysis timeframes. Inference, on the other hand, is performed in real-time and refers to the process of performing classification (or regression) on a set of test input cases using the trained model. In our work, we focus on improving inference latency in order to achieve small analysis timeframes.

Traditional processors running inference algorithms cannot meet the required timing constraints in most cases due to the large overhead of communication protocols, memory accesses, and static (often generic) architectures. Data must first be moved from sensors to CPU buffers, typically by using serial ports, thus limiting the bandwidth. For CPU-based implementations, processing individual test cases results in a large number of cache misses. This is because MLPs have virtually no data reuse for the same test case, and batch processing is not possible due to latency bounds. Moreover, meaningful model sizes tend to be larger than the higher cache levels (closer to the CPU) and hence entire models cannot fit in the L1 or L2 cache. For GPU implementations, further memory transactions are required to move data to and from the device memory over the PCIe bus, which increases the overhead. Low data reuse and batch-less processing also hurt GPU performance since the computations may not be sufficiently parallel to fully utilize the thousands of available cores. For cores that are assigned work, a significant number of cycles are likely to be idle as threads wait for off-chip data to be accessed. ASIC based designs have typically managed to fill this gap by providing massive amounts of resources and specialized pipelines that are tailored for Deep Neural Networks (DNNs); one example is the Google TPU [19]. However, as the number of diverse applications and their associated models grows, these ASICs effectively address a domain, rather than a particular application, and hence are unable utilize application-specific optimizations at the level needed.

Reconfigurable architectures, such as FPGAs, are becoming increasingly popular since their logic can be configured to construct application-specific architectures [20–25]. For MLPs in particular, arbitrary sized models can be easily implemented, scaled, and even transferred to newer generation technologies by recompiling the design. Like GPUs, FPGAs are Commercial-Off-The-Shelf (COTS), which means users can buy the device, generate logic for their desired model, and ensure that the resulting compute pipelines are optimal not only for MLPs, but also for their specific MLP application. Moreover, since the underlying computation structure does not change, complex HDL codes do not have to be written. Instead, simple scripts can be used to create the required

architecture. As FPGA on-chip memory grows, reaching several megabytes in the current generation, users can move all model parameters to the on-chip SRAM and remove idle cycles caused by fetching weights from off-chip DRAM. Moreover, FPGAs offer support for most common serial and parallel protocols and can thus further minimize the latency of memory and I/O transactions by supplying data directly (from sensors) to compute pipelines. All these features result in a transition of the computation from memory bound to compute bound.

Improving latency (and hence performance) of compute-bound MLP inference requires all pipelines to operate both stall free and at high bandwidth. The former is achieved by ensuring modules in the design source/sink data at rates that are constrained by the latter. Since modules in MLP architectures are tightly coupled and operate within the same clock domain, constraints can occur even for indirect connectivity. Therefore, by selecting higher interface bandwidths for particular ports, the complexity (and hence latency) of potentially multiple modules can become significantly large. This increase in complexity can outweigh the benefits of higher module throughput and thus increase the overall latency of the computation.

FPGA-based MLP implementations have received significantly less attention than other DNNs such as Convolutional Neural Networks. The most prominent design is Microsoft's FPGA-based inference architecture, Brainwave [26], targeting low compute-to-data ratio DNN applications such as MLPs, Long Short-Term Memories (LSTMs), and Gated Recurrent Units (GRUs). The memory bandwidth bound is alleviated by using on-chip block RAM for weight matrix storage. For an 8-bit integer implementation on Stratix V FPGAs, Brainwave achieves 2.0 TOps/s, while on Stratix 10 FPGAs, they claim to have 31 TOps/s performance running at 500 MHz. In [27, 28], the authors proposed FPGA-based MLP architectures, but their work serves as a proof-of-concept and is also constrained by off-chip memory bandwidth. Sharma et al. presented an automated DNN design generation framework, DNNWeaver [29], which also depends on DRAM access speed for performance. Moreover, the DNNWeaver compute units are constrained to Digital Signal Processor (DSP)-only implementations and logic cells are not used for ALUs. Gomperts et al. introduce a general purpose architecture for MLP based on FPGAs [30]. Their design generates individual processing elements for each layer, which is not feasible for large neural networks where resources on a single FPGA may not even be sufficient to compute a single layer in parallel.

With regard to stand-alone operation with a direct interface to sensors, FPGAs have shown support for various forms of connectivity without host support. Apart from General Purpose I/O pins [31, 32] that can implement virtually any communication protocol, they also offer direct chip-chip connectivity through Multi-Gigabit Transceivers (MGTs) and network connectivity through dedicated Ethernet controllers. MGTs are serial links that provide low-latency, high-bandwidth, and low-energy interfaces [33–38] and enable communication at rates of 100 Gbps per MGT. Current high-end FPGAs can have close to 100 MGTs. These MGTs can be used to connect multiple FPGAs together and perform computations for models that are too large for a single device. Large multi-FPGA clusters have long been a staple of computational finance and certain other industries. George et al. [39] presented an academic example of a 64 FPGA cluster with direct FPGA-to-FPGA links in a 3D torus network. Similarly, Microsoft's original Catapult System [40, 41] consists of 1632 nodes with FPGAs directly interconnected via MGTs in a series of 6×8 tori.

In our work, we explore the application-aware optimization space for compute-bound MLP inference processors using our proposed FPGA-based architecture. By identifying modules in the critical path and their interconnectivity, we determine and optimize parameters for a given application in order to minimize the latency of the entire model evaluation and meet real-time constraints. This strategy enables our design to be feasible for applications such as real-time medical diagnosis.

Methods

Aim

In this study, we have designed and implemented a real-time Multi-Layer Perceptron inference processor for medical diagnosis using FPGAs. Our focus is on achieving ultra-low analysis latency to meet real-time constraints. The proposed system consists of a modular approach with standardized interfaces that enables hardware of individual functions to be easily modified based on changes to the inference model, design practices, or resource availability. We demonstrate that the ability to be application specific enables FPGA-based designs to choose architecture parameters that minimize latency and maximize utilization of computing resources. Using mass spectrometry benchmarks for cancer detection, we show that FPGAs can outperform traditional computing technologies such as CPUs and GPUs for real-time MLP applications.

Hardware specifications

We have tested our designs on the Intel Arria 10 FPGA (AX115H3F34E2SGE3), which has 427,200 Adaptive Logic Modules (ALMs), 1518 DSP blocks, 53 megabits of on-chip memory, and a maximum of 624 user General Purpose Inputs/Outputs (GPIOs) [42]. The GPU used is a Tesla P100 which has 3594 CUDA cores and a 12 gigabyte High Bandwidth Memory (HBM2) with a peak bandwidth of 549 GB/s. For the baseline, we use an eight-core 2.6 GHz Intel Xeon E5-2650v2 CPU.

Software specifications

FPGA designs are implemented by using Altera OpenCL 16.0.2 [43] to ensure that standard optimizations are applied and the impact of varying design parameters on latency can be fairly determined. Resource usage is measured by using the Quartus Prime Pro 16.0 compiler. GPU reference designs are compiled by using TensorFlow [44] r1.4, python 3.6.2, cuDNN 6.0 and CUDA 8.0. CPU code is also compiled by using TensorFlow r1.4. Both GPU and CPU designs use single-precision floating point, while the FPGA implementation uses fixed-point arithmetic, with arbitrary variable sizes that are optimized for each computation stage.

Benchmarks

We demonstrate the effectiveness of the FPGA-based MLP inference processor by evaluating models for detecting cancer using protein profiles. The two datasets used are obtained from [45] and use Surface-Enhanced Laser Desorption and Ionization (SELDI) protein mass spectrometry to generate data for ion intensities at 15154 mass/charge values. The first is Ovarian-8-7-02 (Ovarian), which contains 91 normal and 162 cancer cases. The second is Prostate Cancer Studies (JNCI), which contains 69 cancer and 63 normal cases. Cancer and normal data for both benchmarks are further divided into training and test sets, as shown in Table 1. For both benchmarks, we propose a model with two hidden layers, with the output determining whether the patient is normal or has cancer. We train the proposed models in single-precision floating point using Tensorflow and find that prediction accuracy for both benchmarks is $> 90\%$. For the FPGA implementation, we convert trained parameters to 8 bits for weights and 32 bits for biases. This conversion is achieved by scaling them, based on a given layer's maximum and minimum values, in order to utilize the entire integer range.

Multi layer perceptrons

In this section, we provide an overview of Multi-Layer Perceptron based neural networks using both logical and computational models. Multi-Layer Perceptron models are typically composed of an input layer containing feature values measured using sensors, an output layer containing the diagnosis result, and, potentially, multiple hidden layers that perform the required computations on the input data. Each layer consists of one or more neurons, depending on the model. MLPs are fully connected as illustrated in Fig. 1a: a neuron in any given layer is connected to the outputs of all neurons in the previous layer.

Table 1 Proposed MLP models for mass spectrometry benchmarks

Model	Dimensions	Training cases	Test cases	Accuracy	F_1 Score
Ovarian	**15154** × 512 × 512 × **2**	160	92	98.9%	0.99
JNCI	**15154** × 64 × 512 × **2**	100	32	90.6%	0.92

The boldface represent the sizes of input and output layers of the Multi Layer Perceptron

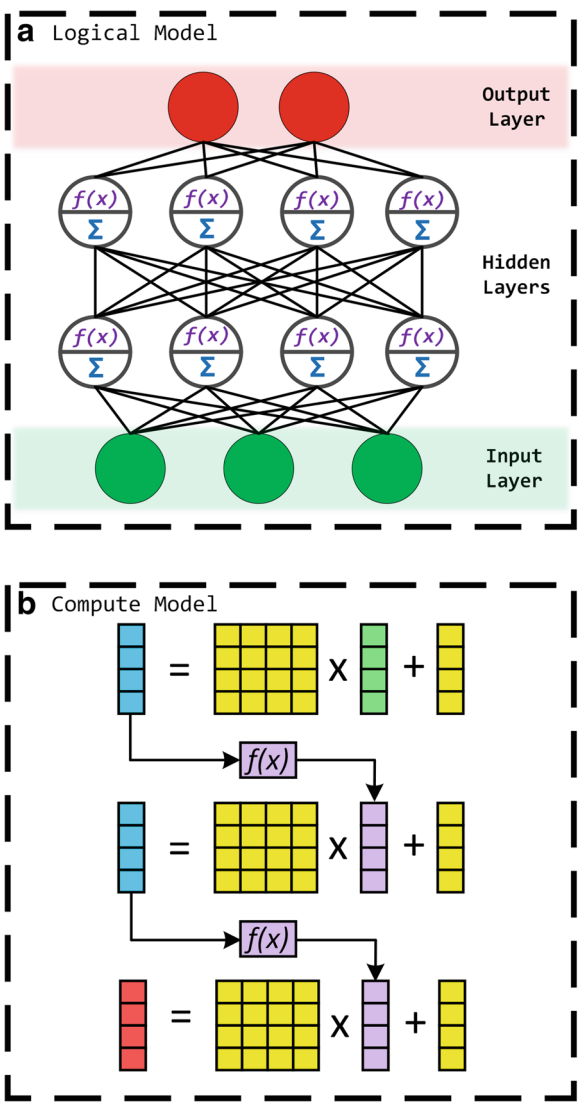

Fig. 1 MLP models for (**a**) logical configuration and (**b**) compute. MLP models can have many layers, with each layer composed of potentially multiple neurons. Each neuron is connected to all neurons in the layer before it. The weights assigned to input connections of a single neuron are stored as a row vector, and every neuron has its own associated row vector. Combining row vectors of neurons in a layer forms the weight matrix for that particular layer. A non-linear function is applied to the weighted sum of a given neuron's inputs, and the result forms the output of the neuron

Inputs to the hidden and output layer neurons are scaled and accumulated by using weights (connection strengths) that are determined during training. A non-linear function, called the activation function, is applied to the result, which then becomes the neuron output. Figure 1b shows this operation represented as a Matrix Vector Multiplication (MVM). Each layer has a unique weight matrix, which contains connection strengths, and a bias vector. Layer inputs are the result vector from the previous layer, or the input vector if it is the first hidden layer. A given row from the $X \times Y$ weight matrix for a given layer represents the connection strengths of Y neurons in the previous layer assigned to one of the X neurons in the current layer. The activation function is applied to individual elements of the output vector. During the training process, all data are floating point. Classification can be performed by using integer arithmetic without loss of accuracy.

MLP Inference architecture

In this section, we present the MLP inference architecture illustrated in Fig. 2. We use a modular approach to component design that enables parameters to be varied in order to implement the optimal dimensions for a given application. Compute and control planes are segregated, enabling additions, deletions, and updates to be easily performed. Individual components within each plane also have well-defined boundaries and interfaces to ensure that design changes can be performed at very high granularity, with minimal effort, and without necessitating changes to other logic beyond required parameter updates. Layers are processed sequentially, with modules performing all computations for a given layer before evaluating the next one.

Compute

Scalar product

For this module, users can specify both the number of Scalar Product evaluations and their size. Computations are performed by using 8-bit variables, while results are accumulated into 32-bit outputs. To minimize latency, we use tree-based structures, rather than systolic arrays, for implementing the Scalar Product modules. This approach ensures that we can scale well to larger input vectors. Typical FPGA implementations focus primarily on DSP-based resources for this stage. However, we provide users with the capability of selecting arbitrary numbers of DSP and ALM-based multipliers at the granularity of a Multiply-Add module as shown in Fig. 3. Based on board resources, users can specify the number of available DSPs while the remaining computing entities are synthesized with ALMs.

Accumulate

If the size of a Scalar Product module is smaller than the input vector, multiple iterations are required to accumulate partial sums and obtain a final value. Moreover, a bias value must be added to the sum. The Accumulate module performs all of these functions by using dedicated *Accumulate Registers*. It receives triggers from the control plane on whether to accumulate or to re-initialize the registers for a new operation cycle.

Activation & requantization

The Activation & Quantization module reads data from the buffer, performs 32-bit ReLU activation ($RELU(x) = max(x, 0)$), and then quantizes data back to 8 bits for the next layer. Quantization is performed by using truncation

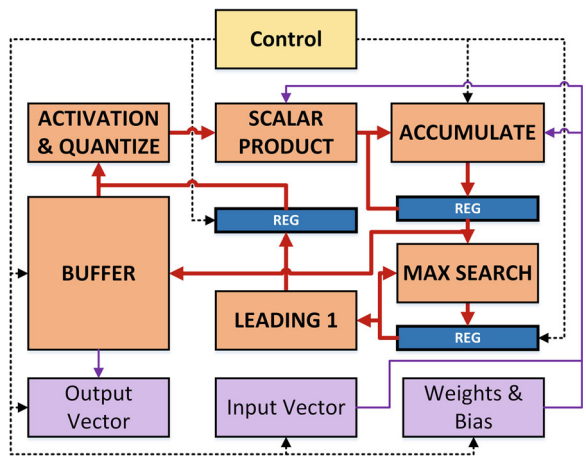

Fig. 2 The architecture of the proposed low latency MLP inference system. The modular-centric approach and well-defined boundaries/interfaces enable updates, additions, and deletions to be performed easily and with minimum changes to adjacent components

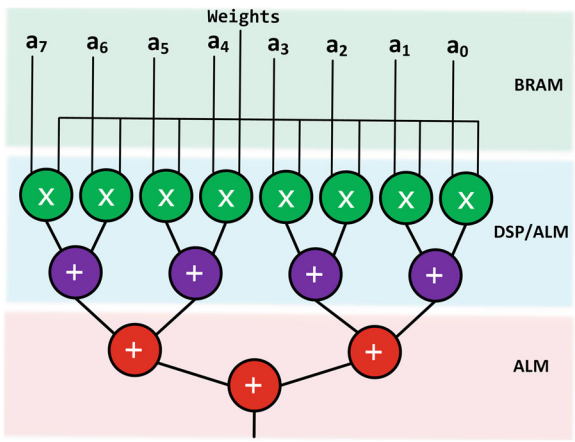

Fig. 3 Scalar product module design using both DSP and ALM multipliers. Each scalar product slice is composed of DSP or ALM multipliers and an adder tree with logarithmic complexity to minimize latency. For DSP-based slices, two multipliers and one adder are implemented in DSP blocks while the remaining adder tree is always implemented by using ALMs

because of the high costs of division hardware. Because of the nature of the operation (compression), the difference in results is small in this particular context. Moreover, ReLU activation ensures that the Most Significant Bit (MSB) of our 8-bit result is always 0. Therefore the effective compression target is 7 bits, which further reduces the difference between division and truncation results.

Max search

Being able to perform quantization requires knowledge of the upper and lower data limits. Because of the ReLU activation, we are guaranteed a lower limit of 0. Searching for the upper limit must be done without stalling the data stream. Consequently, we use the Max Search module to perform local maxima searches on data as it becomes available and update an associated register if a local maximum exceeds the current global maximum. This approach ensures that latency is based on the dimensions of the accumulator outputs and not the full input vector. Employing a tree-based search further reduces the delay.

Leading 1

Once the maximum value for a given output has been determined, we use the Leading 1 module to find the most significant non-zero bit and use this position to perform truncations for quantization. The output is constrained to be between 6 and 30 (on a scale of 0-31) since the former means all values are already within 8 bits while the latter represents the largest possible positive numbers. As with Scalar Product and Max Search, the evaluation is performed with logarithmic complexity.

Buffer

The Buffer module stores result vectors for both the current and the previous layer (input to current layer). While the Buffer is used purely as a memory resource, it is included in our compute plane because of its tight coupling with the Accumulate and Activation & Quantize modules. It is implemented by using registers in order to meet throughput demands for architecture-specific source and sink sizes. A two-bank architecture, Fig. 4, comprising of separate input and output memory banks is used. The output memory bank stores results from the previous layer and supplies this data to the Activation & Requantization module. On the other hand, the input memory bank stores results of ongoing computations by sinking data from the Accumulate Registers. A single-cycle data transfer, from the input bank to the output one, is triggered once all neuron outputs have been computed and the system is processing the global maximum. This ensures that data are ready to be supplied when the next layer is picked up for processing.

Critical path

Of the compute plane modules discussed above, all but the Buffer lie in the critical path. We divide these modules into two categories; the Variable Critical Path (VCP) and the Persistent Critical Path (PCP). The Variable Critical Path is entirely the Scalar Product module. It is equal to the number of calls needed for the Scalar Product module to perform all multiplication operations. Since it is based on the relative dimensions of the weight matrix and Scalar Product module, it will vary for each layer. Once the last set of results has been produced, modules that need to evaluate this last result before a new layer can be processed are referred to as the Persistent Critical Path. It corresponds to a fixed number of cycles independent of layer dimensions. Applicable modules include Accumulator (with register), Max Search (with register), Leading 1, and Activation & Quantize.

Design parameters

Because of the tight coupling of our system design and the latency requirement, certain modules should be able to sink stall-free the entire throughput of their preceding component, as well as source the required throughput to their subsequent one. A chain of such modules where this throughput cannot be modified specifies a design

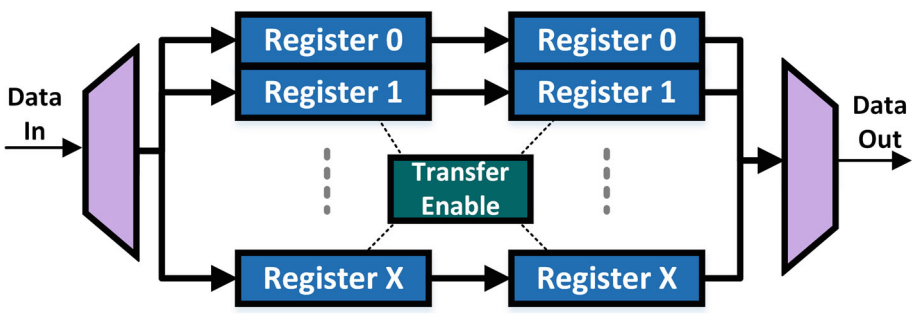

Fig. 4 Structure of the Buffer Module. The two-stage design enables us to hold values of a previous layer (input vector to Scalar Product) while also storing results of current layer computations

parameter. In the proposed architecture, two such chains exist.

The first chain connects the Buffer and Scalar Product modules. It constrains i) the output size of the Buffer, ii) the size of the Activation & Quantize module, and iii) the length of a vector input to the Scalar Product module. We refer to this parameter as *M*. The value of M determines the number of columns being processed in the weight matrix. The second chain connects the Scalar Product to the Max Search module. It constrains i) the number of Scalar Product units, ii) the number of parallel accumulators needed to add partial sums, and iii) the number of elements over which to perform a maximum value search. We refer to this parameter as *N*. The value of N determines the number of rows being processed in the weight matrix. A third chain, between Max Search and Leading 1 modules, always has a throughput of one 32-bit element per cycle and hence does not have a variable design parameter. In our work, we explore the impact of varying M and N on performance.

Latency model

We present an average-latency model for the proposed architecture. We assume standard design practices for implementing the three predominant types of entities: multipliers, pipeline stages, and trees. Equation 1 gives a generic model for latency that can be used to describe combinations of the above. *A* is a multiplicative factor representing the number of pipeline stages per tree layer. *B* refers to the number of variables reduced by the tree while the logarithm computes its corresponding depth. *C* is a constant latency offset representing the minimum cycles taken by any module to perform its assigned computation. *D* represents multiple module calls and is applicable only to modules in the Variable Critical Path, i.e., the Scalar Product multiplication stage.

$$module_i = D_i \left(A_i \lceil log_2(B_i) \rceil + C_i\right) \ cycles \tag{1}$$

Based on the general model above, we determine the total system latency for a given application. The motivation here is to determine the values for M and N that minimize the overall latency, rather than that of a particular layer. As shown in Eq. 2, the total latency is a linear combination of all latencies across all *K* layers of the application. We assume that the Accumulate and Activation & Quantize operations occur in a single cycle (as there are no data dependencies between vector elements and simple computation stages). Max Value Search and Leading 1 are both modeled as trees, but the latter one has constant latency due to a fixed input size (32 bits). We separate the Scalar Product module into multiplication and accumulation stages. The former is modeled as a single-stage pipeline that processes M×N blocks of a weight matrix per cycle. On the other hand, the accumulation stage is absorbed into the Persistent Critical Path and has logarithmic latency based on M.

$$L_{Total} = \sum_{i=1}^{K} \left\{ \left\lceil \frac{rows_i}{N} \right\rceil * \left\lceil \frac{cols_i}{M} \right\rceil + \lceil log_2(M) \rceil + (log_2(1)+1) + 1 + \lceil log_2(N) \rceil + 1 + log_2(32) \right\} + \sum_{i=1}^{K-1} \{log_2(1)+1\} \tag{2}$$

By comparing the effective latencies of Variable (VCP) (Eq. 3) and Persistent (PCP) (Eq. 4) Critical Paths, we demonstrate the substantial impact of design parameters on performance. For larger values of M and N, the latency of VCP decreases, and fewer iterations are needed to perform all required multiplications for a given layer. However, this also causes the latency of PCP to increase and hence can potentially decrease performance despite there being more compute resources. On the other hand, having small values of M and N can still increase overall latency since a greater number of cycles are spent computing scalar products. These complex interactions highlight the need for reconfigurable architectures that can be tuned for a particular application.

$$L_{VCPTotal} = \sum_{i=1}^{K} \lceil rows_i/N \rceil * \lceil cols_i/M \rceil \tag{3}$$

$$L_{PCPTotal} = K(\lceil log(M) \rceil + \lceil log(N) \rceil + 8) - 1 \tag{4}$$

Control

Using the modules and their functions outlined previously, Fig. 5 gives an overview of the algorithm for performing inference on a given test vector. The control unit is responsible for generating event triggers to coordinate the flow of data between different modules (green). These triggers are based on system states (blue) and include start/end indicators, variable updates (e.g., resets, increments, swap), read/write signals, and data source selection (e.g., the buffer module, external on-chip memory).

A detailed implementation of the control unit is shown in Fig. 6. For given values of M and N and application model dimensions, we can determine how long each layer will take, at which cycle individual triggers will be given, and the computation being performed by each module at any given time. All these can be coordinated based on a single global counter (Main Counter). By having an instruction-free implementation, the overhead of

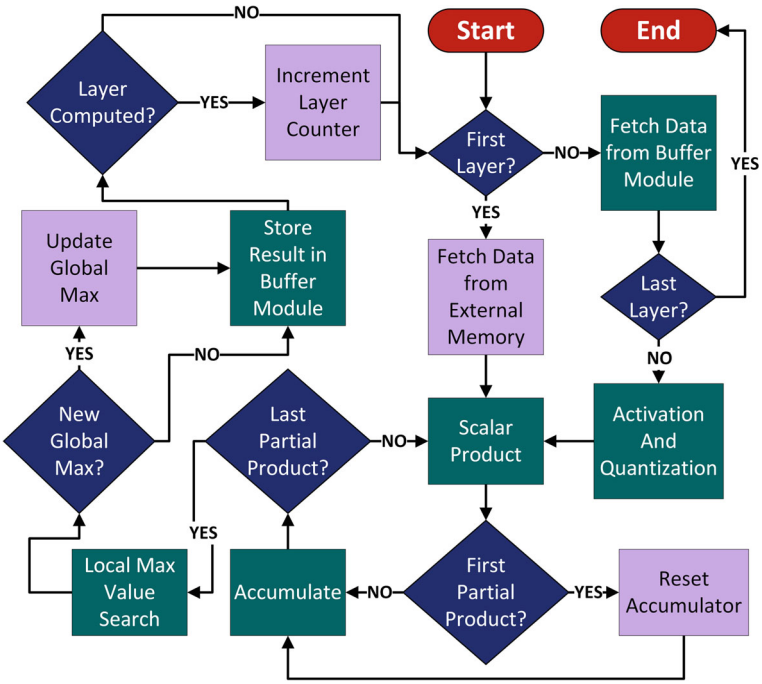

Fig. 5 Algorithm: An overview of the algorithm used to determine the event triggers needed to control the flow of data in the inference processor

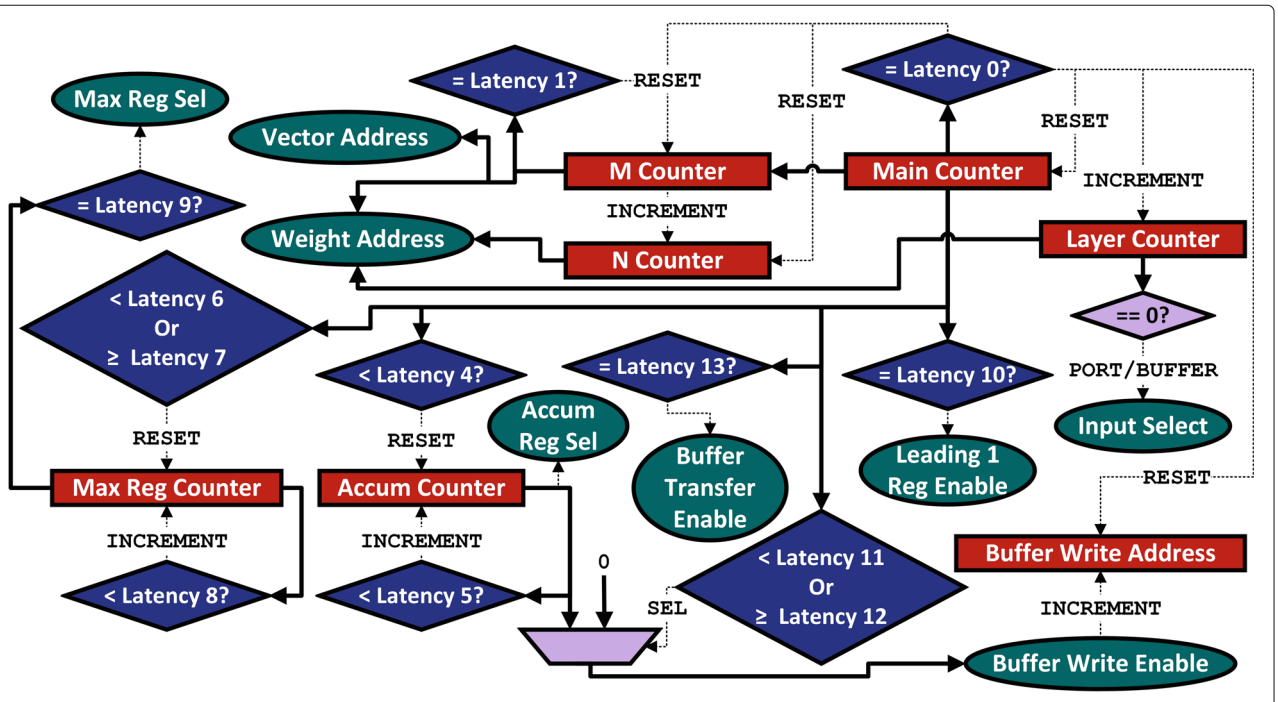

Fig. 6 Control Unit: Execution of the entire application model can be done based on a single global counter without any feedback from modules or user-supplied run-time instructions. Values of the latency tags for each layer are hard-coded into the system and selected based on the layer counter value

Table 2 Latency tags for defining trigger ranges

Tag	Value
Latency_0	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N) + 1 + L_LO
Latency_1	MBLOCKS[i]
Latency_4	L_QA(M) + L_MM(M,N)
Latency_5	MBLOCKS[i] -1
Latency_6	L_QA(M) + L_MM(M,N) + L_AC(N) + 1 + L_MX(N)
Latency_7	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N)
Latency_8	MBLOCKS-1
Latency_9	MBLOCKS-1
Latency_10	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N) + 1 + L_LO -1
Latency_11	L_QA(M) + L_MM(M,N) + L_AC(N) + 1
Latency_12	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1
Latency_13	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1

fetching and decoding instructions is avoided, and end-to-end data flow for the entire application can be made stall free.

To define ranges and trigger points for setting and resetting values of control signals and state machine counters, we use latency tags. Each tag is based on the latency of an individual module in the corresponding data path. Table 2 lists these tags and their values. QA, MM, AC, MX, and LO refer to latencies of the Activation & Quantization, Scalar Product, Accumulate, Max Value Search, and Leading 1 modules, respectively. Constants represent latencies of registers at the output of the Accumulate and Max Value Search Modules. MBLOCKS and NBLOCKS refer to the number of blocks the weight matrix of a layer can be divided into in each dimension, while BLOCKS is the product of these, that is, the number of cycles needed for the entire weight matrix of a layer to be processed. Tags 2 and 3 are reserved for external connectivity in future work.

Results

Measured latency

Figure 7 shows the latency of the Scalar Product module. Data points for (M ← 8:256 , N ← 8), (M ← 8 , N ← 8:256), (M ← 16, N ← 16), (M ← 32, N ← 32) and (M ← 64, N ← 64) are measured values while the remaining points are estimated based on observed trends. As illustrated by the graph, larger M values correspond to higher latency, while the latency due to N is mostly constant (due to simple design replication).

Figure 8 illustrates the latency of modules in the Persistent Critical Path (except for Scalar Product accumulation). As was demonstrated with our system model previously, most modules have latency offsets based on pipeline depth and thus have nearly invariant latencies with respect to their associated parameter. With regard to Max Value Search, however, we get very large latencies despite it being a tree-based implementation. This is because of the resource overhead of a signed comparator as compared with a simple adder (≈2x more ALMs per comparator based on synthesis results).

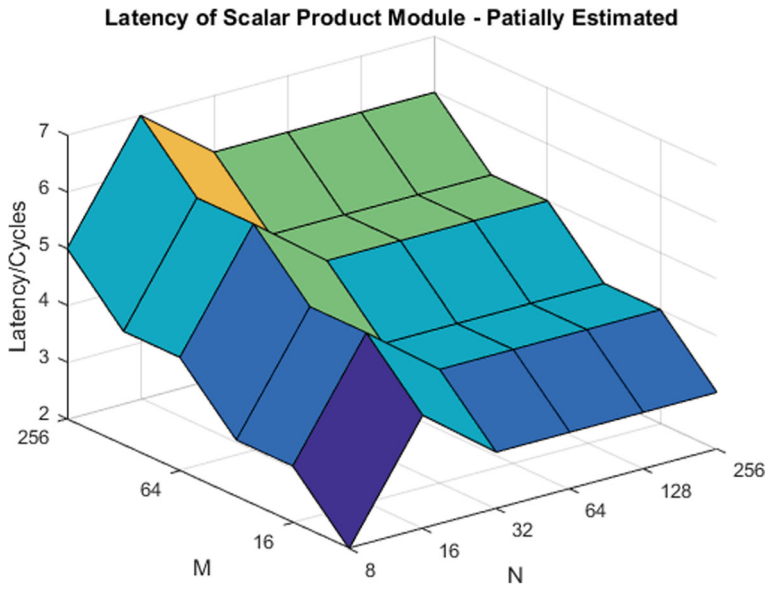

Fig. 7 Latency comparison of various M and N for Scalar Product module. Latency increases with larger M (more tree stages) but is invariant in N (tree replication)

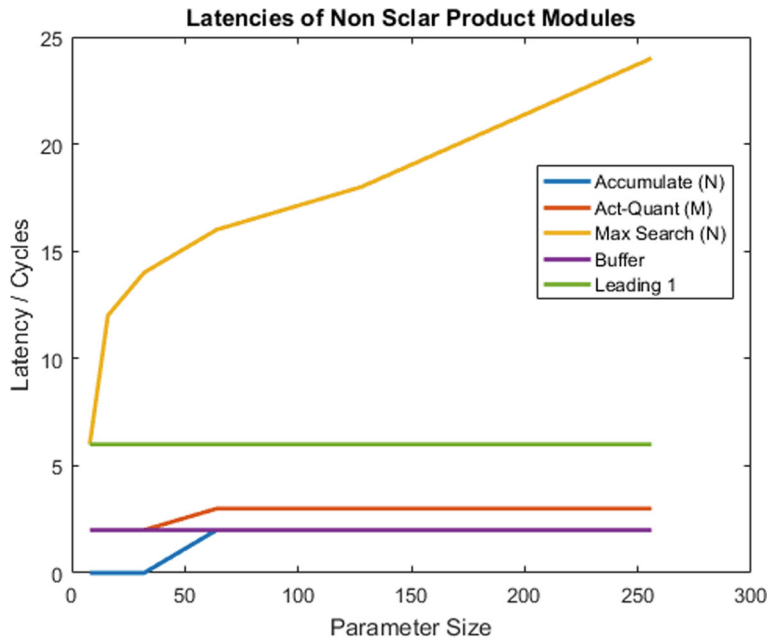

Fig. 8 Latency of critical path modules based on their constraining parameter

Figure 9 shows the total latency of our system for a single iteration of all modules. We observe that having larger values of M, instead of N, reduces latency by 20% on average.

Resource usage

We have tested our designs on the Altera Arria-10AX115 FPGA. Table 3 gives the post-fit resource usage of the processor. We compile the design with $M = 256$ and $N = 8$. These values minimize overall inference latency based on the dimensions of benchmark models, Eq. 2, and latency results from Fig. 9. From the resource usage, we see that the design occupies less than half the chip. Therefore, based on Eq. 2, either a larger value of M can be used to further reduce latency, or a second (independent) inference processor can be included.

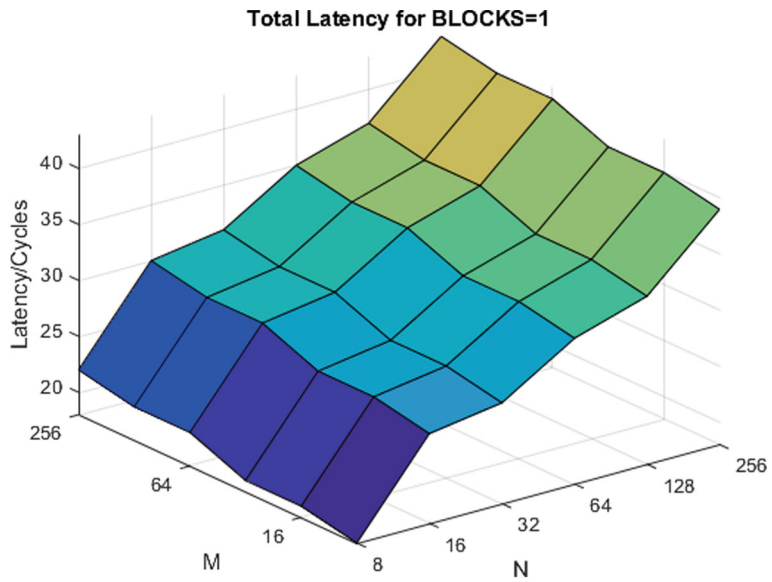

Fig. 9 Total latency of the system for a single iteration. Having a larger value of M results in significantly lower latency than larger values of N

Table 3 FPGA implementation details

M	N	ALM	DSP	Frequency
256	8	57008/427200 (13%)	512/1518 (34%)	295 MHz

Performance

We compare the performance of the FPGA implementation with an eight-core 2.6 GHz Intel Xeon E5-2650v2 CPU and an NVidia Tesla P100 GPU. Execution time for a single input test case is measured by performing inference for batch sizes shown in Table 1 and then taking the average. From the results (Fig. 10), we see that the FPGA outperforms the CPU and high-end GPU for both benchmarks, despite only using half the chip resources. FPGA execution time is 56x and 372x faster than CPU and 4x and 112x faster than GPU for the Ovarian and JNCI benchmarks, respectively.

Discussion

The hurdle to performing real-time analysis on patient data is meeting timing constraints. This is difficult to do on traditional, fixed-logic platforms because of a large number of sources of overhead. Meeting timing constraints with fixed-logic platforms may continue to become less feasible. With technological advancements and greater integration of sensors into medical procedures, both the data throughput of individual sensors and their overall number can be expected to increase. Our results highlight the importance of application-specific optimizations for minimizing the latency of MLP inference and meeting significantly stricter performance requirements.

When constructing modules, several factors must be considered when determining parameter sizing, such as the impact on VCP and PCP. For PCP, when standard optimizations were applied using OpenCL-based architecture generation, the size and complexity of basic components, along with the nature of the overall computation, had a significant impact on the module latency. This is nearly invariant of the application itself and depends more on the design strategies for implementing individual modules. The VCP, on the other hand, is significantly more application-specific since it depends on the ratio of parameter values to nearly every layer size in the model. Thus, while there is an initial bias toward increasing the size of M over N, all latency contributions must be considered on a per-application basis to determine the optimal architecture.

An important observed trend in the performance results emphasizes the impact of batch-less execution on traditional accelerators. As the size of the model decreases, from Ovarian to JNCI, the FPGA execution time is also reduced since fewer computations are required for the smaller first hidden layer in JNCI. However, the GPU performance for the same transition shows a decrease. The reason is that GPUs depend heavily on batch processing of multiple test vectors in order to get good utilization of the thousands of computing cores. As a result, the lower number of test vectors for JNCI results in worse GPU performance despite there being fewer overall computations. Overall, we expect the CPU and GPU performance values to be even lower if batch-less test sets are run (as opposed to computing the average).

Conclusion

We show the importance of application-specific optimizations in order to minimize latency and maximize resource utilization for MLP inference. By directly interfacing with and processing sensor data during procedures, FPGAs can perform real-time analysis and provide diagnostic feedback that can be critical to achieving higher

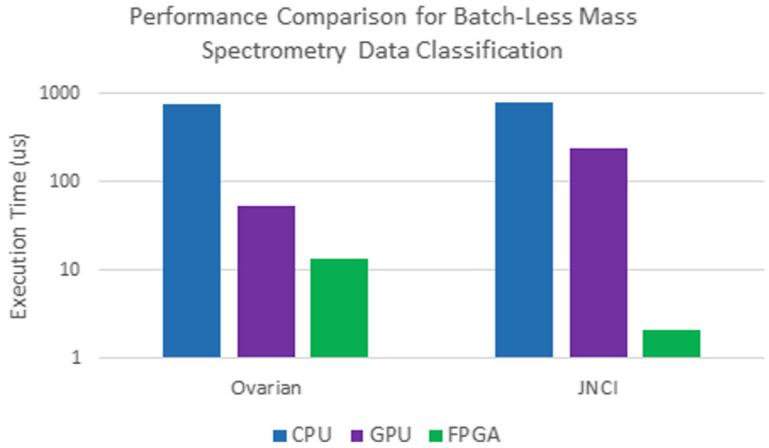

Fig. 10 Performance Comparison between CPU, GPU, and FPGA. FPGA outperforms both the CPU and GPU with average speedups of 144x and 21x, respectively

percentages of successful patient outcomes. We propose a modular architecture that enables modifications to be easily performed based on the application model, design updates, and resource availability as the design is migrated to different FPGAs. We demonstrate that correct parameter sizing, based on the application, can reduce latency by 20% on average. Further, we show that in an application with tightly coupled data-path and latency constraints, having a large amount of computing resources can actually reduce performance. Moreover, since the FPGA does not require batch processing of inputs, it can operate with ultra-low latency in order to meet real-time constraints. Using mass-spectrometry benchmarks, our proposed FPGA design outperforms both CPU and GPU implementations with average speedups of 144x and 21x,respectively.

Abbreviations

ALM: Adaptive logic module; ALU: Arithmetic logic unit; BCI: Brain-computer interface; COTS: Commercial-off-the-shelf; CADx: Computer-aided diagnosis; DNN: Deep neural network; DSP: Digital signal processor; FPGA: Field programmable gate arrays; GRU: Gated recurrent unit: GPIO: General purpose input/output; GPU: Graphics processing unit; HBM: High bandwidth memory; LSTM: Long short-term memory; ML: Machine learning; MVM: Matrix vector multiplication; MSB: Most significant bit; MGTs: Multi-gigabit transceivers; MLP: Multi-layer perceptron; PCP: Persistent critical path; RF: Rotation forest; SVM-RFE: Support vector machine recursive feature elimination; SELDI: Surface-enhanced laser desorption and ionization; VCP: Variable critical path

Acknowledgements

This work is supported in part by the National Science Foundation through Awards CNS-1405695 and CCF-1618303/7960; by a grant from Red Hat; by Altera through donated FPGAs, tools, and IP; and by Gidel through discounted FPGA boards and daughter cards. This research used the resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DoE) Office of Science User Facility supported under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. The authors would also like to thank Hafsah Shahzad for valuable comments and suggestions to improve the quality of this paper.

Funding

Publication costs were funded by a grant from Microsoft.

Availability of data and materials

The datasets generated and/or analyzed during the current study are available in the FDA-NCI Clinical Proteomics Program Databank.

https://home.ccr.cancer.gov/ncifdaproteomics/ppatterns.asp

About this supplement

This article has been published as part of *BMC Bioinformatics Volume 19 Supplement 18, 2018: Selected Articles from the Computational Approaches for Cancer at SC17 workshop*. The full contents of the supplement are available online at https://bmcbioinformatics.biomedcentral.com/articles/supplements/volume-19-supplement-18.

Authors' contributions

AS and CY developed the proposed architecture. YA and KZ provided the work on MLP applications and datasets. MH contributed to all aspects of the work including background, architecture design and benchmarking. All authors have read and approved of the final manuscript.

Ethics approval and consent to participate

Not Applicable.

Consent for publication

Not Applicable.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Author details

[1]Computer Architecture and Automated Design Lab, Boston University, Boston, MA, USA. [2]Argonne Leadership Computing Facility, Argonne National Laboratory, Lemont, IL, USA. [3]Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, USA.

Published: 21 December 2018

References

1. CANDLE. Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer. http://candle.cels.anl.gov. Accessed 24 Sep 2018.
2. Balaprakash P, Alexeev Y, Mickelson SA, Leyffer S, Jacob R, Craig A. Machine-learning-based load balancing for Community Ice CodE component in CESM. USA: Springer; 2014, pp. 79–91.
3. Sen F, Hills S, Kinaci A, Narayanan B, Davis M, Gray S, Sankaranarayanan S, Chan M. Combining First Principles Modeling, Experimental Inputs, and Machine Learning for Nanocatalysts Design. Bull Am Phys Soc. 2017;62. http://meetings.aps.org/link/BAPS.2017.MAR.A1.1.
4. Kononenko I. Machine Learning for Medical Diagnosis: History, State of the Art and Perspective. Artif Intell Med. 2001;23(1):89–109.
5. Vallabhaneni A, Wang T, He B. Brain—Computer Interface. In: Neural Engineering. New York: Springer; 2005. p. 85–121.
6. Müller K-R, Tangermann M, Dornhege G, Krauledat M, Curio G, Blankertz B. Machine Learning for Real-Time Single-Trial EEG-Analysis: From Brain–Computer Interfacing to Mental State Monitoring. J Neurosci Methods. 2008;167(1):82–90.
7. Shoeb A, Guttag J. Application of Machine Learning to Epileptic Seizure Detection. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10). Madison: Omnipress; 2010. p. 975–982.
8. Müller H, Michoux N, Bandon D, Geissbuhler A. A Review of Content-Based Image Retrieval Systems in Medical Applications—Clinical Benefits and Future Directions. Int J Med Inform. 2004;73(1):1–23.
9. Wernick M, Yang Y, Brankov J, Yourganov G, Strother S. Machine Learning in Medical Imaging. IEEE Signal Proc Mag. 2010;27(4):25–38.
10. Rahman MM, Bhattacharya P, Desai B. A Framework for Medical Image Retrieval using Machine Learning and Statistical Similarity Matching Techniques with Relevance Feedback. IEEE Trans Inf Technol Biomed. 2007;11(1):58–69.
11. Wang Y, Fan Y, Bhatt P, Davatzikos C. High-Dimensional Pattern Regression using Machine Learning: From Medical Images to Continuous Clinical Variables. Neuroimage. 2010;50(4):1519–35.
12. Suzuki K. Pixel-Based Machine Learning in Medical Imaging. J Biomed Imaging. 2012;2012:1.
13. Zacharaki E, Wang S, Chawla S, Yoo DS, Wolf R, Melhem E, Davatzikos C. Classification of Brain Tumor Type and Grade using MRI Texture and Shape in a Machine Learning Scheme. Magn Reson Med. 2009;62(6):1609–18.
14. Pereira F, Mitchell T, Botvinick M. Machine Learning Classifiers and fMRI: A Tutorial Overview. Neuroimage. 2009;45(1):199–209.
15. Mazurowski M, Habas P, Zurada J, Lo J, Baker J, Tourassi G. Training Neural Network Classifiers for Medical Decision Making: The Effects of Imbalanced Datasets on Classification Performance. Neural Netw. 2008;21(2-3):427–36.
16. Suzuki K. Machine Learning in Computer-Aided Diagnosis: Medical Imaging Intelligence and Analysis: Medical Imaging Intelligence and Analysis. Hershey: IGI Global; 2012.
17. Ozcift A, Gulten A. Classifier Ensemble Construction with Rotation Forest to Improve Medical Diagnosis Performance of Machine Learning Algorithms. Comput Methods Prog Biomed. 2011;104(3):443–451.
18. Li M, Zhou Z-H. IEEE Trans Syst Man Cybern Syst Hum. 2007;37(6):1088–98.
19. Jouppi NP, Young C, Patil N, Patterson D, Agrawal G, Bajwa R, Bates S, Bhatia S, Boden N, Borchers A, Boyle R, Cantin P-l, Chao C, Clark C,

Coriell J, Daley M, Dau M, Dean J, Gelb B, Ghaemmaghami TV, Gottipati R, Gulland W, Hagmann R, Ho CR, Hogberg D, Hu J, Hundt R, Hurt D, Ibarz J, Jaffey A, Jaworski A, Kaplan A, Khaitan H, Killebrew D, Koch A, Kumar N, Lacy S, Laudon J, Law J, Le D, Leary C, Liu Z, Lucke K, Lundin A, MacKean G, Maggiore A, Mahony M, Miller K, Nagarajan R, Narayanaswami R, Ni R, Nix K, Norrie T, Omernick M, Penukonda N, Phelps A, Ross J, Ross M, Salek A, Samadiani E, Severn C, Sizikov G, Snelham M, Souter J, Steinberg D, Swing A, Tan M, Thorson G, Tian B, Toma H, Tuttle E, Vasudevan V, Walter R, Wang W, Wilcox E, Yoon DH. In-Datacenter Performance Analysis of a Tensor Processing Unit. SIGARCH Comput Archit News. 2017;45(2):1–12. https://doi.org/10.1145/3140659.3080246.

20. Sanaullah A, Khoshparvar A, Herbordt MC. FPGA-Accelerated Particle-Grid Mapping. In: International Symposium on Field-Programmable Custom Computing Machines. Piscataway: IEEE; 2016. p. 192–195.
21. Xiong Q, Herbordt MC. Bonded Force Computations on FPGAs. In: Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium On. Piscataway: IEEE; 2017. p. 72–75.
22. Mahram A, Herbordt MC. NCBI BLASTP on High-Performance Reconfigurable Computing Systems. ACM Trans Reconfigurable Technol Syst (TRETS). 2015;7(4):33.
23. Chiu M, Herbordt MC. Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems. ACM Trans Reconfigurable Technol Syst. 2010;3(4):23.
24. Sukhwani B, Herbordt MC. FPGA Acceleration of Rigid Molecule Docking Codes. IET Comput Digit Tech. 2010;4(3):184–195.
25. VanCourt T, Herbordt MC. Families of FPGA-based algorithms for approximate string matching. In: International Conference on Application Specific Systems, Architectures, and Processors. Piscataway: IEEE; 2004. p. 354–364.
26. Chung E, Fowers J, Ovtcharov K, Papamichael M, Caulfield A, Massengil T, Burger D, et al. Accelerating Persistent Neural Networks at Datacenter Scale. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave. Accessed 24 Sep 2018.
27. Panicker M, Babu C. Efficient FPGA Implementation of Sigmoid and Bipolar Sigmoid Activation Functions for Multilayer Perceptrons. IOSR J Eng. 2012;2:1352–6.
28. Taright Y, Hubin M. FPGA Implementation of a Multilayer Perceptron Neural Network using VHDL. Signal Process (ICSP), IEEE Int Conf. 1998;2:1311–4.
29. Sharma H, Park J, Mahajan D, Amaro E, Kim JK, Shao C, Mishra A, Esmaeilzadeh H. From High-level Deep Neural Models to FPGAs. Microarchitecture (MICRO). Piscataway: IEEE; 2016, pp. 1–12.
30. Gomperts A, Ukil A, Zurfluh F. Development and Implementation of Parameterized FPGA-based General Purpose Neural Networks for Online Applications. Ind Inform IEEE Trans. 2011;7(1):78–89.
31. Latino C, Moreno-Armendariz MA, Hagan M. Realizing General MLP Networks with Minimal FPGA Resources. In: 2009 International Joint Conference on Neural Networks. 2009. p. 1722–1729. https://doi.org/10.1109/IJCNN.2009.5178680.
32. Zhai X, Ali AAS, Amira A, Bensaali F. MLP Neural Network Based Gas Classification System on Zynq SoC. IEEE Access. 2016;4:8138–46. https://doi.org/10.1109/ACCESS.2016.2619181.
33. Sheng J, Humphries B, Zhang H, Herbordt MC. Design of 3D FFTs with FPGA clusters. In: High Performance Extreme Computing Conference (HPEC), 2014 IEEE. Piscataway: IEEE; 2014. p. 1–6.
34. Altera. Altera Transceiver PHY IP Core User Guide. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/archives/ug-01080-1.7.pdf. Accessed 24 Sep 2018.
35. Altera. Arria 10 Device Overview. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf. Accessed 24 Sep 2018.
36. Sheng J, Yang C, Herbordt M. Towards Low-Lantency Communication on FPGA Clusters with 3D FFT Case Study. In: Proc. Highly Efficient and Reconfigurable Technologies. New York: ACM; 2015.
37. Sheng J, Xiong Q, Yang C, Herbordt MC. Collective Communication on FPGA Clusters with Static Scheduling. ACM SIGARCH Comput Archit News. 2017;44(4):2–7.
38. Yang C, Sheng J, Patel R, Sanaullah A, Sachdeva V, Herbordt M. OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. High Performance Extreme Computing Conference (HPEC). Piscataway: IEEE; 2017.
39. George AD, Herbordt MC, Lam H, Lawande AG, Sheng J, Yang C. Novo-G#: Large-scale Reconfigurable Computing with Direct and Programmable Interconnects. High Performance Extreme Computing Conference (HPEC). Piscataway: IEEE; 2016, pp. 1–7.
40. Putnam A, Caulfield AM, Chung ES, Chiou D, Constantinides K, Demme J, Esmaeilzadeh H, Fowers J, Gopal GP, Gray J, et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. ACM SIGARCH Comp Archit News. 2014;42(3):13–24.
41. Sheng J, Yang C, Sanaullah A, Papamichael M, Caulfield A, Herbordt MC. HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In: Field Programmable Logic and Applications (FPL), 2017 27th International Conference On. Piscataway: IEEE; 2017. p. 1–4.
42. Intel Arria 10 Product Table. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf. Accessed 24 Sep 2018.
43. Altera. Intel FPGA SDK for OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. Accessed 24 Sep 2018.
44. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker P, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X. TensorFlow: A System for Large-scale Machine Learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16. Berkeley: USENIX Association; 2016. p. 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899.
45. Biomarker Profiling, Discovery and Identification. home.ccr.cancer.gov/ncifdaproteomics/ppatterns.asp. Accessed 20 Feb 2018.

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com