Neural Networks as Function Primitives: Software/Hardware Support with X-FILES/DANA

Schuyler Eldridge, Tommy Unger, Marcia Sahaya Louis, Jonathan Appavoo, and Ajay Joshi
Boston University
{schuye, tommyu, marcia93, jappavoo, joshi}@bu.edu

Amos Waterland and Margo Seltzer
Harvard University
apw@seas.harvard.edu, margo@eecs.harvard.edu

***Abstract*—Neural networks and machine learning provide utility for analyzing, inferring, learning, and predicting information from data. Substantial interest in this area has created a wide proliferation of different software and hardware implementations. As neural network usage assumedly becomes a dominant component in future applications, this wide proliferation of software and hardware begins to become a problem. With this work, we treat neural network computation as a functional primitive and explore the consequences of this approach for hardware design and software management.**

I. Introduction

Neural networks (NNs) are graphs composed of layers of neurons that mimic the basic structure and processing of the brain. Each layer in the graph processes, in parallel, data received from the previous layer to generate inputs for the next layer. In this way, information follows a *feedforward* path from the input layer, through a number of hidden layers, to an output layer. Each neuron computes a weighted operation on its inputs (e.g., a weighted sum) and applies an activation function that determines its output. Generally, this activation function is a continuous, differentiable threshold function (e.g., a sigmoid or an inverse tangent) that mimics the fire/no-fire behavior of biological neurons.

NNs provide good solutions for classification (mapping inputs to discrete classes) or regression tasks (mapping inputs to continuous outputs) where precise solutions are unknown or hard to find. This trait hinges on the selection of suitable values for connection weights. NN weights can *learn* or be *trained* through an incremental update method like gradient descent. During learning, connection weights are updated to minimize a cost function defined at the output, like mean squared error (MSE) of an output with respect to a known, correct output. Differentiable activation functions enable efficient gradient computation of each neuron with respect to the output cost function via *backpropagation* of errors through the NN. Weights can then be updated across the gradient to minimize the output cost function for a specific input–output pair or for all known input–output combinations.

While a number of existing software [4] and hardware [2], [3] implementations exist, usually without hardware learning support, current interest in NNs has created a proliferation of different software and hardware implementations. While the former (e.g., Caffe, Theano) can target multiple backends (e.g., CPUs and GPUs), proliferation of custom NN hardware implementations creates issues of incompatibility. With this work, we take a step back and view NN computation as a common computational kernel and treat NN computation as a first class functional primitive. We then view NN hardware accelerators as a tightly-integrated coprocessor of general purpose microprocessors analogous to vector or floating point units. Under this treatment, we present published and ongoing work on the development of user and supervisor software and hardware that enables the management of NN transactions initiated by processes in a modern multi-process operating system (OS) [1]. Our software and ISA infrastructure takes the form of a set of **Ex**tensions **f**or the **I**ntegration of Machine **L**earning in **E**veryday **S**ystems (X-FILES). As an example accelerator, we use a backend **D**ynamically **A**llocated **N**eural Network **A**ccelerator (DANA). We additionally demonstrate the usage of this combined X-FILES/DANA system.

II. X-FILES/DANA: Software and Hardware

At the user level, the X-FILES encompass an API for NN *transactions*. An NN transaction encapsulates a request by a user process to access a specific NN, the communication of inputs, backend accelerator processing, and the return of outputs. We describe an NN with an NN configuration—a binary representation of all the information necessary to execute a specific NN. Others treat this as a program with instructions that move data amongst distinct arithmetic units [2]. We opt for a raw description of the NN to enable more finely grained, run-time allocation of resources. At the supervisor level, the X-FILES enable the safe management of multiple transactions from disparate processes and for the definition of sets of shared NN configurations. These sets—address spaces—enable the safe sharing of NN configurations between processes. Transactions are identified with `TID`s, address spaces with `ASID`s, and NN configurations with `NNID`s.

A. Software

Table I shows the user and supervisor API for the X-FILES. A user process initiates a transaction, writes input data, and performs a blocking read to initiate and close an NN transaction:

```
nnid = 1;
tid = newWriteRequest(nnid, 0, num_output);
write_data(tid, inputs[0], num_input);
read_data_spinlock(tid, &outputs, num_output);
```

This simple structure requires significant backend support. Dereferencing of an `NNID` must happen in a safe way that

TABLE I. X-FILES SOFTWARE LIBRARY FUNCTIONS FOR COMMUNICATION WITH THE X-FILES ARBITER AND MANAGING THE ASID–NNID TABLE

Function	User/Supervisor	Description
`tid = newWriteRequest(nnid, learningType, numOutputs)`	user	Initiate a new transaction returning a TID
`writeData(tid, *inputs, num_input)`	user	For register mode, write input data and, optionally, training data
`readDataSpinlock(tid, *output, num_output)`	user	For register mode, try to read output data until successful
`tidKill(tid)`	user	Kills an executing transaction
`oldTid = setAsid(asid)`	supervisor	Change the ASID returning the old TID to the OS for storage
`setAntp(*table)`	supervisor	Set the ASID–NNID Table Pointer and the number of ASIDs
`asidNnidTableCreate(**table, numAsid, numNnid)`	supervisor	ASID–NNID Table constructor
`asidNnidTableDestroy(**table)`	supervisor	ASID–NNID Table destructor
`nnid = addNnid(**table, asid, *nnConfiguration)`	supervisor	Adds an NN Configuration to an existing ASID–NNID Table
`removeNnid(**table, nnid)`	supervisor	Remove a specific NNID from an exsiting ASID–NNID Table

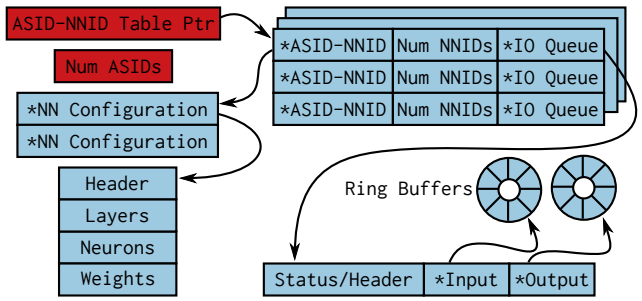

Fig. 1. An ASID–NNID Table for dereferencing NN configurations

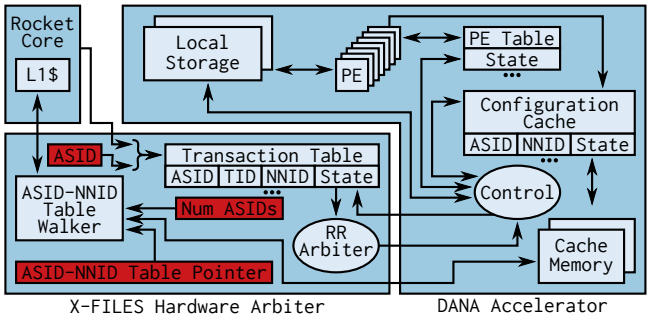

Fig. 2. The X-FILES/DANA hardware architecture. Neural network transactions are managed by a hardware arbiter and executed on a backend accelerator.

aligns with our address space model. The OS manages an ASID–NNID Table which dereferences an NN configuration from an ASID and NNID. Figure 1 shows the data structure used to represent an ASID–NNID Table. Given an ASID–NNID Table Pointer (ANTP), a supervisor system can discern the set of NNIDs, assigned sequentially, that belong to a specific ASID. Each NN configuration, shown at a high level in the bottom left of Figure 1, consists of four regions that completely describe an NN: global information stored in a header and a description of the layers, neurons, and all weights. We additionally provide an asynchronous communication method via input and output in-memory ring buffers. OS creation of an ASID–NNID Table can then be accomplished with the following code snippet:

```
asid_nnid_table * table;
asid_nnid_table_create(&table, num_asid,
    num_configs);
attach_nn_configuration(&table, asid, file_nn);
set_antp(table);
```

B. Hardware

The hardware for our X-FILES/DANA system spans two distinct components—an X-FILES hardware arbiter (storing transaction and ISA state) and a backend accelerator DANA—shown in Figure 2. The hardware arbiter maintains the state of all executing NN transactions and facilitates communication with the ANTP via an ASID–NNID Table Walker. DANA consists of a number of processing elements (PEs) which implement the underlying computations of a neuron. NN configurations are cached locally on DANA while intermediate computations are stored in local per-transaction storage.

Execution of NN computations on DANA then involves the mapping of the neurons, described by the NN configuration, to PEs. DANA allocates these PE resources dynamically allowing for the computations of multiple NN transactions to be interleaved and enabling higher overall throughput of NN computation.

III. OPEN SOURCE AND FUTURE WORK

We are currently in the process of open sourcing the X-FILES software libraries (with Linux kernel integration) as well as X-FILES/DANA hardware designs. This ongoing work treats X-FILES/DANA hardware as a drop-in accelerator of the RISC-V rocket microprocessor [5] providing fast integration with an existing hardware and software ecosystem.

ACKNOWLEDGMENTS

This work was supported by a NASA Space Technology Research Fellowship, the National Science Foundation with a Graduate Research Fellowship under Fellow ID 2012116808, a Google Faculty Research Award, and a CAREER award under ID CNS-1254029 and CNS-1439069.

REFERENCES

[1] S. Eldridge, A. Waterland *et al.*, "Towards general-purpose neural network computing," in *Proc. PACT*, 2015.

[2] H. Esmaeilzadeh, A. Sampson *et al.*, "Neural acceleration for general-purpose approximate programs," in *Proc. MICRO*, 2012.

[3] C. Farabet, C. Couprie *et al.*, "Learning hierarchical features for scene labeling," *IEEE Tran. Pattern Anal.*, vol. 35, no. 8, pp. 1915–1929, 2013.

[4] Y. Jia, E. Shelhamer *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[5] A. Waterman, Y. Lee *et al.*, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014.