Towards General-Purpose Neural Network Computing

Schuyler Eldridge,* Jonathan Appavoo,† and Ajay Joshi*
Department of Electrical and Computer Engineering
†*Department of Computer Science*
Boston University
Boston, MA
{*schuye, jappavoo, joshi*}*@bu.edu*

Amos Waterland and Margo Seltzer
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA
apw@seas.harvard.edu
margo@eecs.harvard.edu

***Abstract*—Machine learning is becoming pervasive; decades of research in neural network computation is now being leveraged to learn patterns in data and perform computations that are difficult to express using standard programming approaches. Recent work has demonstrated that custom hardware accelerators for neural network processing can outperform software implementations in both performance and power consumption. However, there is neither an agreed-upon interface to neural network accelerators nor a consensus on neural network hardware implementations. We present a generic set of software/hardware extensions, X-FILES, that allow for the general-purpose integration of feedforward and feedback neural network computation in applications. The interface is independent of the network type, configuration, and implementation. Using these proposed extensions, we demonstrate and evaluate an example dynamically allocated, multi-context neural network accelerator architecture, DANA. We show that the combination of X-FILES and our hardware prototype, DANA, enables generic support and increased throughput for neural-network-based computation in multi-threaded scenarios.**

I. Introduction

Neural Networks (NN) and machine learning techniques are effective computing methods for some the most challenging sets of applications including character recognition [1], stock market prediction [2], dynamic branch prediction [3], automation of compiler optimization [4], and image processing [5]. Success has prompted a wide array of NN research including:

1) Specialized hardware architectures that improve the performance and energy efficiency of machine learning techniques [5]–[17]
2) Schemes for extending the exploitation of NN processing to applications that are not explicitly programmed to use them, e.g., approximate computing [18]–[23] or auto-parallelization [24]

These diverse implementations and usage cases drive fascinating innovation. As diversity increases, however, a gap is developing between these innovations and the state of today's hardware and software.[1] As such it is worth considering how generalized NN processing can be explicitly integrated into today's hardware and software environment.

Explicit integration of NN computing starts with a precisely specified hardware-software interface. Such an interface allows NN hardware diversity to expand while, at the same time, encouraging a diverse set of software use cases to be explored. From a hardware perspective, one would like to see the continued exploration of the full spectrum of NN accelerator technologies from dedicated NN digital logic units [5]–[13], [18]–[23] to biologically inspired analog/sub-threshold implementations [14]–[17]. Similarly, from a software perspective, one wants to encourage both explicit and implicit usage models to grow. In the former, explicit case, one wants to encourage applications and libraries to easily utilize NN facilities in a fine-grained fashion with portability across hardware technologies. In the later, implicit case, compilers and runtime systems should, also portably, exploit available hardware facilities to transparently use NN processing as needed, e.g., stitching in NNs in place of function calls to enact approximations [18]–[20] or exploiting NNs at runtime to model an application's behavior and enable parallel speculations [24].

To date, hardware NN technologies have developed predominately in the context of a static and dedicated application use scenario—for the most part a single NN is being used by a single application to do a single computation at a time. In contrast, multiprocessing and fine-grained threading characterize today's general purpose computing landscape. While the hardware/software interface need not be predicated on a multi-context, concurrent accelerator, such an interface would be of limited *future* utility if not properly designed to support such accelerators.

Towards these goals of both generality and simultaneous multiprocessing, we provide a clean slate approach to NN computation broken into two distinct contributions—a set of software/hardware extensions and a new NN accelerator architecture:

The X-FILES Hardware/Software Extensions: We define software/hardware E**x**tensions **f**or the **I**ntegration of Machine **L**earning in **E**veryday **S**ystems, or **X-FILES**, that provide a standard approach to describe

[1]While we tend to shy away from off-putting hubristic analogies, we believe the current NN hardware landscape bears similar opportunities to the state of floating point before William Kahan and the IEEE 754 effort—accepted as beneficial, but fragmented.

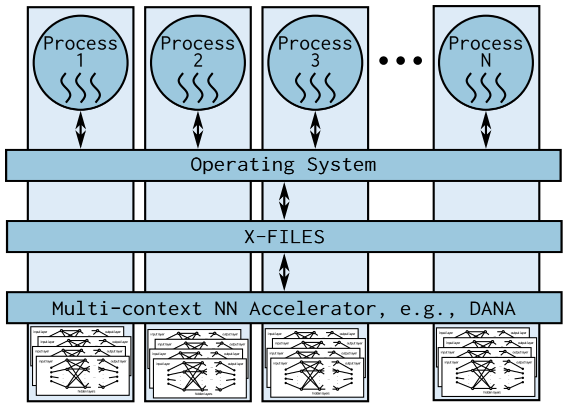

Figure 1. A multiprocessing system has processes composed of threads managed by an operating system. We propose a set of hardware/software extensions, the X-FILES, that enable the multi-context management of requests by processes to execute NNs in their respective address spaces on an underlying NN accelerator.

NN computation and interface software with underlying NN acceleration hardware.[2] The X-FILES treat the unit of NN computation as a *transaction*, i.e., a request by a process to compute the output of a network for an input. The *X-FILES software stack*—user applications and user libraries with backing supervisor/operating system (OS) code—communicate user transactions to an *X-FILES hardware arbiter* that offloads units of NN computations to an underlying NN accelerator. Using the X-FILES, 1) each application can construct and use multiple NNs, 2) each NN can have arbitrary all-to-all connectivity in its structure, including unique input and output dimensions, and 3) an application can issue many concurrent requests for processing using its NNs. To concretize and realistically constrain the X-FILES extensions we develop and present the X-FILES as using the RoCC accelerator interface of a RISC-V microprocessor [25]–[28].

The DANA Accelerator Architecture: We propose a new **D**ynamically **A**llocated **N**eural Network **A**ccelerator architecture, **DANA**, as an example X-FILES accelerator backend. Differing from existing accelerators [5]–[23], DANA targets concurrent execution of similar or dissimilar NN computations. To achieve this, DANA and the X-FILES arbiter work together to schedule and overlap NN computation on available DANA resources—a collection of processing elements, cached NN configurations, and intermediate storage. DANA's multiprocessing capabilities support NN computation from multiple threads/cores resulting in higher accelerator throughput.

[2]We apologize for any confusion related to the choice of 1990s television-inspired acronyms. There is no affiliation with "files" or "filesystems." Also, the acronym *is* plural and can be replaced grammatically with "extensions."

Figure 1 illustrates our overall model and how the X-FILES and DANA fit together. Processes, isolated in memory and composed of one or more threads, share underlying computer hardware by means of an OS. Each thread uses one or more NNs in its overarching application or to enable an alternative computing paradigm, e.g., NN-backed approximate computing [18]–[23] or microprocessor state prediction [24]. The X-FILES software extensions provide both a user-level API for NN computation and necessary OS data structures and management functions to allow safe sharing of NN configurations. The X-FILES hardware arbiter provides an interface for user and system software extensions to manage and execute NN computations on a backend NN accelerator.

II. X-FILES: Software/Hardware Extensions for NN Computation

Fundamentally, the X-FILES software/hardware extensions manage NN configurations and transactions in disjoint address spaces as defined below:

Neural Network Configuration – a concise hardware and software-friendly binary description of an NN's structure. Each configuration is uniquely identified with an OS-managed *NN Identifier* (`NNID`).

Neural Network Transaction – the basic unit of NN computation. A transaction encapsulates a request by a user process to use a specific `NNID`, the initial communication of inputs, execution on an NN accelerator, and final communication of outputs. A transaction may optionally involve learning. Each NN Transaction is identified by a *Transaction Identifier* (`TID`) assigned by hardware.

Address Space – the set of NN configurations and other data structures shared between a group of processes. Each Address Space is identified by an OS-assigned *Address Space Identifier* (`ASID`).

The X-FILES are, therefore, software/hardware extensions that manage `TID`s belonging to `ASID`s that define sets of allowable `NNID`s. This management comes in the form of user and supervisor software that directly communicates with a hardware arbiter. The arbiter maintains the state of all executing `TID`s. A backend accelerator (of which we provide DANA as one possibility in Section III) then works with the arbiter to execute transactions. Figure 2 shows this division in the context of a multicore system.

The communication of information about `NNID`s, `TID`s, and `ASID`s between X-FILES software running on one or more connected cores and the hardware arbiter occurs through two possible transfer modes:

Register Transfer – special instructions that send core register data to the accelerator

Memory Transfer – asynchronous transfers through pointers to preestablished data structures

Based on the amount of data that must be transferred, software selects the more efficient mode. For example,

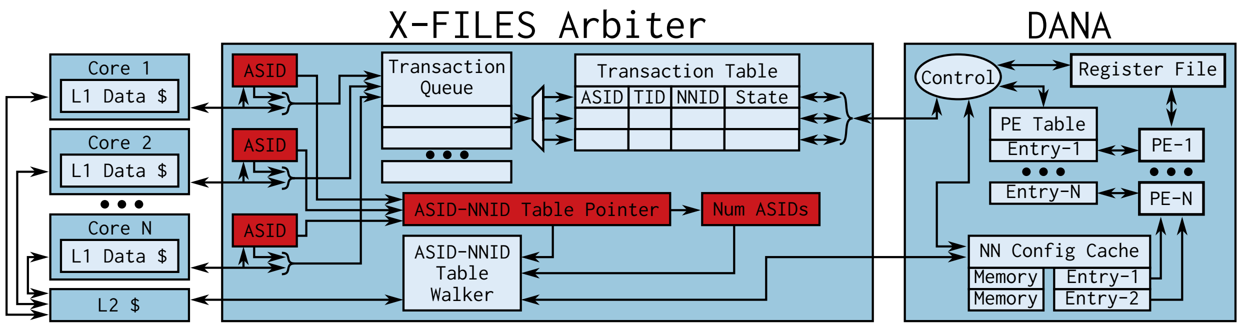

Figure 2. X-FILES hardware arbiter with DANA as a backend accelerator. Registers set by a supervisor (e.g., `ASID`s) are dark red.

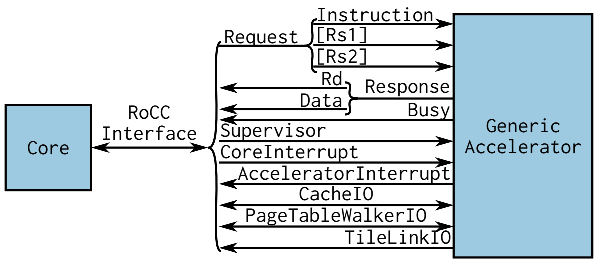

Figure 3. The RoCC accelerator interface [25]–[28]

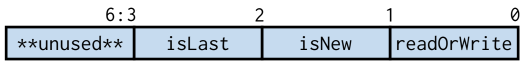

Figure 4. Function field bits that encode the instructions in Table I

Table I
ASSEMBLY INSTRUCTIONS TRANSMITTED OVER THE ROCC INTERFACE

			Instruction	
`isLast`	`isNew`	`readOrWrite`	User	Supervisor
0	1	1	New Transaction	—
0	0	1	Data Write	Set `ANTP`
1	0	1	Last Data Write	—
0	0	0	Data Read	Set `ASID`

we expect an approximate computing application with two inputs and one output to use register transfer. Conversely, we anticipate an image processing application with streaming, one-time-use data to use asynchronous memory transfers. We leave this choice of transfer mode, however, to the developer. The short nature of supervisor requests, outlined in more detail in a following subsection, are best suited to register transfers.

We further subdivide memory transfers into those involving *virtual* or *physical* addresses. While this would normally be a hard, design-time decision (as is common with similar accelerator interfaces [29], [30]) we prefer to not restrict the generality of the X-FILES during definition. Instead, and as with register/memory mode, we take the view that these transfer modes should be exposed to agents (like the hardware designer, library writer, compiler, or OS) that can make the most informed decision regarding interface choice. Consequently, the option to choose between virtual and physical addressing at compile or runtime enables us to choose the mode that is supported by the accelerator and suits the characteristics of the application.

As there already exist a number of transport layer interfaces for dedicated computational accelerators we standardize on one of these instead of defining our own. We select the RoCC interface used by the RISC-V Rocket core [25]–[28], and shown in Figure 3, due to its ready integration with an open source ISA and hardware designs. Note, that while we use a standardized interface to a RISC-V microprocessor, the X-FILES are independent of transport layer and ISA.

The RoCC interface defines a set of communication lines for transmitting/receiving register data, a full ISA instruction, status/exception bits, and direct communication with the memory hierarchy. Using standard RISC-V extensions for RoCC [27], we can then use dedicated instructions to send data to/from the X-FILES arbiter. We currently use four instructions, shown in Table I. Three bits of the instruction "function" field, shown in Figure 4, define the type of communication—`readOrWrite` denotes whether this is a read or a write, `isNew` indicates that this is a new transaction request, and `isLast` specifies that this is the end of a data stream.

The X-FILES register mode uses these instructions directly. In memory mode, the arbiter accesses memory on dedicated lines with *virtual* or *physical* addresses previously communicated using register mode. In the virtual case, address translation occurs either locally on the arbiter (through an arbiter TLB and page table walker) or remotely using the core's memory management unit (MMU).

A. X-FILES User Software

The primary user software functions, shown in the top portion of Table II, enable user processes to access NN computation resources:

- **`newWriteRequest`** – initiate a new NN transaction for a specific `NNID`. The user process receives a `TID` which it can then use to reference this transaction.

Table II
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, type)`	user	Initiate a new transaction of specific `type`, returning a `TID`
`writeData(tid, *inputs, *expectedOutput)`	user	For register mode, write input data and, optionally, training data
`readDataPoll(tid, *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(antp, numAsids)`	supervisor	Set the `ASID–NNID` Table Pointer and the number of `ASIDs`
`asidNnidTableCreate(**table)`	supervisor	`ASID–NNID` Table constructor
`asidNnidTableDestroy(**table)`	supervisor	`ASID–NNID` Table destructor
`nnid = addNnid(**table, *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

- **`writeData`** – referencing a `TID`, transmit inputs and expected outputs (in the case of a transaction involving learning) to NN hardware. In memory transfer mode, this function can be circumvented with direct writes to an IO ring buffer.
- **`readDataPoll`** – repeatedly try to read output data for a `TID` until successful. As with **`writeData`**, this function is optional in memory transfer mode.

Thereby, a complete NN transaction from the perspective of a user process consists of a sequence of these three instructions. We additionally define the **`tidKill`** instruction that allows a process to kill a `TID` in its address space.

B. X-FILES Supervisor Software

Supervisor software—libraries incorporated in a modified OS kernel—enables safe, multi-context access to NN acceleration hardware and NN configurations. The RoCC interface uses a supervisor bit that is set when a privileged process is running. The value of this bit causes the X-FILES arbiter to interpret the instructions in Table I differently, specifically to modify supervisor registers on the arbiter.

Memory protection of process' data and NN configurations occurs through the intentional OS partitioning of processes wishing to access NN accelerator resources into address spaces. Processes allowed to access each other's data and NN configurations (e.g., a parent and child) live in the same address space and, consequently, have the same `ASID`. Each interface from a core to the X-FILES arbiter has a separate `ASID` register (see Figure 2). The OS sets each of these registers with the **`setAsid`** function based on the underlying address space of the process executing on each core. The X-FILES arbiter then stamps any subsequent user requests from cores with their respective `ASIDs`. Hardware uses these `ASIDs` and `TIDs` (where `TIDs` are generated by the arbiter) to fully disambiguate transactions.

To ensure memory protection of NN configurations we employ a data structure called an `ASID–NNID` Table shown in Figure 5. The OS creates, destroys, and manages (adds or removes NN configurations) the `ASID–NNID` Table with the supervisor functions in Table II. Upon creation of or a change to the `ASID–NNID` Table, the OS uses **`setAntp`** to

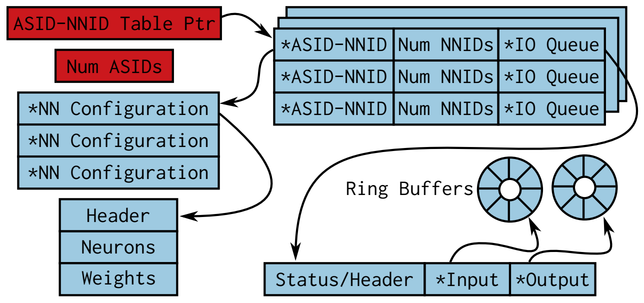

Figure 5. `ASID–NNID` Table used for `NNID` memory protection. An `ASID–NNID` Table Pointer, stored in the X-FILES arbiter and set by the OS, points to *one specific* `ASID–NNID` Table. This `ASID–NNID` Table contains one entry per `ASID` that points to an `NNID` dereference table (where `NNIDs` can be translated to NN Configuration addresses) and an IO Queue (containing some status bits and pointers to IO ring buffers). The X-FILES arbiter also stores the number of `ASIDs` allowing it to detect an invalid `ASID` outside the bounds of the table. The OS assigns `ASIDs` and `NNIDs` sequentially enabling their use for `ASID–NNID` Table indexing.

change the `ASID–NNID` Table Pointer (`ANTP`), an arbiter register that points to the memory location of the `ASID–NNID` Table. After receiving a valid `ANTP`, the arbiter uses this to find an NN configuration in memory. The **`setAntp`** function also sets a register indicating the number of valid `ASIDs`. Due to the sequentially assigned nature of `ASIDs`, the arbiter can use the number of valid `ASIDs` to prevent reading beyond the end of the `ANTP`.

The `ASID–NNID` Table consists of groups of pointers to NN configurations indexed by `ASID` and `NNID`. This rigid structure enables the OS to place hard restrictions on which `NNIDs` are accessible by each `ASID`. Each `ASID` entry also contains a pointer to a set of IO ring buffers. In memory transfer mode, these ring buffers are used to asynchronously transfer data between a core and the arbiter.

C. X-FILES Hardware Arbiter

The X-FILES arbiter (Figure 2) provides an interface between user processes wanting to execute NN transactions and the actual accelerator hardware. When a new transaction arrives, the arbiter enters this in a Transaction Queue—a FIFO structure of not-yet-executing transactions. In the event of the Transaction Queue being full, the arbiter asserts the RoCC interface busy bit indicating that it cannot accept new transactions. The arbiter's Transaction Table pulls transactions from the queue and acts as a record of the state of all transactions executing on the backend NN accelerator. Due to the tight integration of the Transaction Table with the underlying accelerator we discuss this in more detail within the context of DANA in Section III-C.

When the backend accelerator, DANA or similar, indicates that it has free resources to execute transactions, the X-FILES arbiter selects a transaction, communicates the transaction state to the accelerator, and updates the transaction state. When the accelerator needs access to memory resources (primarily to load a specific NN configuration), the X-FILES arbiter uses an `ASID–NNID` Table Walker to ensure memory protection. Inevitable exceptional cases in the arbiter or NN accelerator are communicated through the RoCC interrupt interface back to a core where they are resolved by the OS.

III. DANA: An Accelerator for the X-FILES

The X-FILES hardware/software extensions form only part of a complete system for general-purpose NN computing—the X-FILES need a backend accelerator to execute NN transactions. Since existing NN accelerator architectures [5]–[23] focus on single-processing of transactions, we developed a new architecture better aligned with the X-FILES' multiprocessing capabilities. We view this simultaneous multiprocessing quality as a necessary requirement of future accelerator architectures for two reasons:

1) Application-specific NN computing techniques will require an increasing number of simultaneous NN transactions, e.g., an NN image processing backend with a separate NN per user [5] or microprocessor state prediction using ASC [24]
2) Several concurrent applications will integrate NN computation or adopt NN-backed techniques to improve energy efficiency, e.g., approximate computing [18]–[23]

To meet these requirements, we propose DANA, a **D**ynamically **A**llocated **N**eural Network **A**ccelerator. DANA reads the Transaction Table of the X-FILES arbiter and dynamically assigns logical neurons to fixed Processing Elements (PEs). As the design space of accelerators that interface with the X-FILES is exceedingly large, we make some initial assumptions about DANA:

- DANA uses a binary NN configuration format similar to that used by FANN [31].
- DANA's underlying computational units support only one type of NN flavor, multilayer perceptrons (MLPs), with 32-bit fixed point arithmetic and all-to-all NN topologies.
- DANA supports only feedforward NN computation.

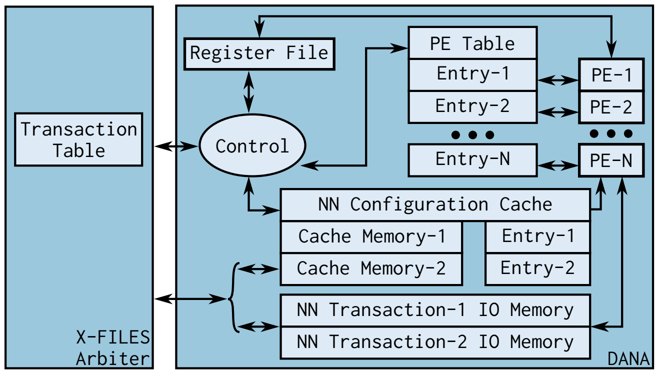

Figure 6. Architecture of DANA. Control logic dynamically allocates processing elements to neurons of executing NN transactions read from the X-FILES arbiter. Inputs and outputs are stored in per-transaction memories, NN configurations are stored in a local cache, and intermediate computations are stored in a Register File.

By extending the NN configuration structure and the range of operations supported by the PEs, additional NN flavors can be supported, e.g., Convolutional Neural Networks. DANA's control logic can be extended to update its local NN configuration and thereby enable hardware incremental or batch learning directly in hardware.

DANA's current version consists of five distinct architectural components shown in Figure 6:

- A **Transaction Table**, located inside the X-FILES arbiter (Figure 2), that maintains the state of all currently executing NN transactions (Section III-C)
- A **Configuration Cache** that stores recently used NN Configurations to exploit temporal reuse of specific NNs (Section III-D)
- A **Register File** storing intermediate outputs (Section III-E)
- A number of **Processing Elements** (PEs) that perform the operation of one neuron (Section III-F)
- **Control Logic** that maps logical neurons of transactions to physical PEs and facilitates communication between the X-FILES arbiter and DANA (Section III-G)

A. Example Transaction Execution on DANA

To demonstrate the basic operation of DANA, consider an X-FILES arbiter with one valid NN transaction. This NN transaction is characterized by its `ASID`, `TID`, and `NNID`.

Initially, this transaction is in an unknown state with regards to whether its corresponding configuration (specifying the layer, neuron, and weight information) is loaded in DANA's Configuration Cache. Control logic queries the Configuration Cache to see if the required configuration is available. On a miss, the cache reserves an entry in the Configuration Cache Table—a unit that stores the state of Configuration Cache entries—and requests the missing NN Configuration from the X-FILES arbiter. The X-FILES arbiter then uses the `ASID–NNID` Table Walker to load this configuration through a core's memory hierarchy. Upon receiving this NN Configuration information, the Configuration Cache stores it in a local memory. The cache sends a ready response to the X-FILES arbiter updating this transactions' state to valid, i.e., "ready to be executed." This ready response also contains auxiliary information about global NN configuration, i.e., the total number of layers and neurons, the number of fractional bits used in the fixed point representation, and the cache memory location of (a pointer to) the first NN weight.

DANA then begins processing the first hidden layer of this NN transaction. DANA queries the Configuration Cache for layer-specific information and updates the Transaction Table. Control logic reserves registers in the register file to store the intermediate outputs of this layer (one for each neuron in the layer). Control logic then assigns the first neuron in the NN to an unallocated PE. PE assignment involves setting various flags and fields for that PE as well as communicating the locations of its inputs and where the PE should send its outputs. This information is stored in a PE Table with one entry for each PE. Each cycle, the control logic selects the next neuron in the hidden layer and assigns it to a free PE so long as free PEs exist. In our current DANA implementation, one PE is assigned per cycle due to limited bandwidth between control logic and PEs. With additional control logic and PE reservation stations, multiple PE assignments can occur per cycle.

After allocation, PEs operate independently of DANA's control logic. The PE Table keeps a record of where each PEs inputs are sourced, where its outputs will be sent, and what address it should request from the Configuration Cache for its weights. PE inputs and outputs can be read from or written to private IO memory (for inputs and outputs of the NN transaction) or to registers in the Register File (for intermediate hidden layer outputs). In case multiple PEs need to read inputs/weights or are ready to output data, they use round robin arbitration to decide which PE gets priority. Currently, only one PE has priority, however, this can be extended to two simultaneous accesses by utilizing both ports of dual-ported storage elements. Additional improvements can come from adding multiple memory banks, overclocking the memory, or grouping like accesses from PEs. After receiving inputs and weights, a PE begins executing. After a multi-cycle processing operation, the PE sends its output to the register file or to output IO memory. This process continues until all logical neurons of a transaction have been mapped to PEs. The Transaction Table then marks the transaction as done and its outputs can be returned to the initiating process.

B. Block and Element Organization

DANA operates on wide groups of data, called *blocks*, composed of multiple 32-bit data *elements*, as shown in Figure 7. The choice of block width affects all modules that exchange data. For example, when a PE requests new inputs, that PE is actually requesting a block from IO memory or the Register File. We made this design decision to reduce requests by PEs for input and weight data and is reflected by increased performance for larger block sizes.

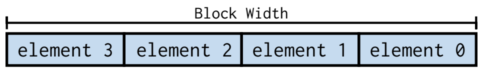

Figure 7. Organization of a *block* with four *elements*.

C. Transaction Table

The Transaction Table stores information about all in-flight NN Transactions. Each entry in the Transaction Table is broken down into *Transaction Status Bits*, global *Transaction Information* (e.g., `TID`, total number of layers), and *Transaction Progress* (e.g., the current layer being processed) as shown in Figure 8.

There are a total of seven *Transaction Status Bits*. The `nnValid`, `nnReserved`, and `nnDone` bits respectively signify whether or not an entry is valid, reserved and waiting for inputs, or if the NN transaction for this entry has completed execution. The `regFileNeedsRegs` bit asserts when this entry needs to reserve register blocks in the Register File where intermediate layer outputs will be stored. Reserved blocks form a non-contiguous linked list, hence, the `regFileNeedsNext` bits assert when this transaction entry needs to know the next reserved register block in the list. The `cacheValid` and `cacheWait` bits are used to determine when the state of the Configuration Cache is valid and if the control logic is waiting on the cache to load a configuration from memory.

Transaction Information fields identify the `ASID`, `TID`, the `NNID`, and configuration parameters for the whole NN, i.e., the fixed point precision binary point and the total number of layers and neurons. The *Transaction Progress* fields define which layer and overall neuron are currently being processed. Additionally, these fields store the current neuron *within the layer* and the total number of neurons in the current layer. Together, the in-layer current neuron field and total neurons field determine when an NN transaction layer is done and when new layer information from the Configuration Cache is needed. Progress fields also store a pointer to the configuration of the current neuron in the Configuration Cache. This pointer is passed to a PE during allocation. Finally, input and output register block indices are maintained.

Each entry in the Transaction Table has private IO memory for inputs and outputs. PEs assigned to process the neurons in the first hidden layer of an NN read their inputs from IO memory. PEs assigned to process the neurons in the last layer of an NN output data to IO memory.

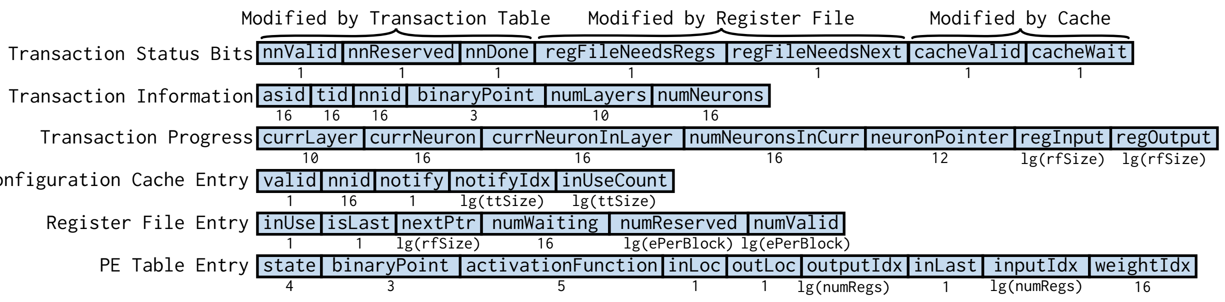

Figure 8. Organization of a Transaction Table entry consisting of Transaction Status Bits, Transaction Information, and Transaction Progress. Also shown are a Configuration Cache entry, a Register File entry, and a Processing Element table entry.

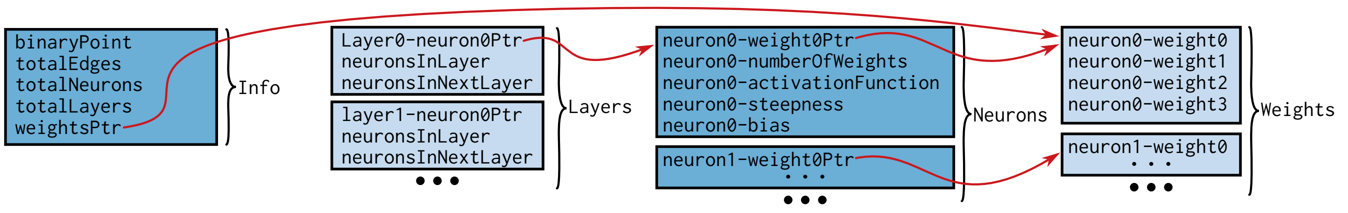

Figure 9. The condensed FANN configuration format consists of four sections: *Info*, containing global NN information, *Layers*, containing layer information and a pointer to the first neuron configuration for that layer, *Neurons*, containing neuron-specific information, and *Weights*, containing all weights for a neuron. All sections and all weights for a given neuron are aligned on block boundaries.

D. *Configuration Cache*

The Configuration Cache stores recently used NN configurations in dedicated per-entry memory. We store NN configurations locally to exploit temporal locality of NN configurations in existing application workloads, e.g., approximate computing [18]–[23] or ASC [24]. Each configuration, shown in Figure 9, is a reduced binary representation of the FANN configuration data structure consisting of four sections: Global Info, Layer Info, Neuron Info, and Weight Info. The global *Info* section contains data applicable to the whole NN and populates the *Transaction Information* part of a Transaction Table entry. The *Layers* section, containing layer specific information and a pointer to the first neuron in the layer, is loaded into the Transaction Table as each layer is processed. The *Neurons* section defines neuron-specific parameters—the number of weights, a pointer to the first weight for this neuron, as well as computation parameters (the activation function, steepness, and bias)—accessed by the PEs. The *Weights* section contains a list of all NN weights and is read by PEs.

An NN configuration is stored using the same parameterized element–block structure described in Section III-B. We align the Info block, first Layer block, first Neuron block, and the start of each weight block on a block boundary to prevent unaligned access overheads. Our alignment approach does incur storage overheads due to empty space arising from forced block alignment.

Each Configuration Cache entry contains state information, shown in Figure 8, and a private storage area for the NN Configuration. State information includes a `valid` bit, an `NNID` field, notification information fields (`notify` and `notifyIdx`) and an `inUseCount` field. On a Configuration Cache miss, a memory or core access is generated with the help of the X-FILES arbiter. We maintain an `InUseCount` field that tracks the number of transactions using a specific cache entry. Only unused cache entries can be evicted from the cache on a Configuration Cache miss. When a request for a specific `NNID` arrives, a cache miss causes the `notify` bit to be set and `notifyIdx` field to record the index of the Transaction Table entry requesting that `NNID`. Once the cache has successfully loaded the requested configuration, it generates a response to DANA's control logic indicating its validity.

While our current version requires that NN configurations fit in one Configuration Cache memory, we plan to support larger NN configurations in a future extension. Specifically, the Configuration Cache will work with the X-FILES arbiter to load a new set of weights from memory whenever needed.

E. *Register File*

DANA uses a Register File to store intermediate outputs. Each entry in the register file contains one block of register elements and some metadata. Register blocks are reserved by DANA's control logic before an NN layer begins execution to ensure that each allocated PE has a dedicated location to write its output. Control logic will consequently not start processing an NN layer until there are enough free register blocks. To accommodate the case when there are more outputs in a layer than registers in a block, the Register File implements a forward linked list. Each Register File entry contains an `isLast` bit that indicates if it is the last in a list as well as a `nextPtr` field that holds the index of the next register block in a list. PEs understand this and will load, as needed, all blocks in a list.

We implement the Register File using a dual-ported memory array. Surrounding combinational logic enables a two-cycle element write (with input data forwarding for the consecutive write case) and one-cycle block read. Register block reads and writes are initiated by PEs independently of the control logic. The Register File returns register blocks and metadata to requesting PEs. The PEs use this metadata to know if this is the last block they need to process, which block they should request next, and what entry is last in a partially filled block. Each Register File entry (see Figure 8) maintains the number of expected reads (`numWaiting`) for a particular block. By enforcing our all-to-all connectivity restriction, we know that this value should initially be set to the number of neurons in the next layer. As the Register File receives PE read requests, this count is decremented. Once the read count for a block reaches zero, the Register File invalidates that entry and DANA's control logic is free to reuse it for other intermediate data storage. In case multiple PEs want to access the register file at the same time, round-robin arbitration is used to choose a PE.

F. Processing Elements (PEs)

Each PE performs one MLP neuron computation—the application of an activation function (AF) to an input–weight inner product:

$$\text{output} = \text{AF}\left[\sum_{\text{All Inputs } i} \text{Input}_i \times \text{Weight}_i\right]$$

DANA's control logic handles PE selection and assignment. During assignment, DANA's control logic provides the selected PE with configuration information, i.e., the number of fixed point fractional bits, the number of input–weight pairs the PE needs to process, and the locations of its inputs, outputs, and weights. After assignment, PEs operate autonomously. PEs request inputs from the Register File or IO memory and weights/neuron-specific information (i.e., activation function, sigmoid steepness, and bias) from the Configuration Cache. If multiple PEs need input and weight data, one is chosen using round robin arbitration. A PE computes a partial input–weight inner product whenever it has a valid block of inputs and weights.

As each multiplication doubles the original bit width, products are truncated to the original fixed point precision by arithmetically right shifting by the value of the binary point. Note that this does introduce a bias towards negative infinity due to the asymmetry of truncation rounding in two's complement. To avoid unnecessary logical complexity from arbitrary shifts, the allowed number of fixed point bits is currently limited to 7–13. We determined these binary point values using the FANN library, which selects a maximum binary point value that will not generate fixed point accumulator overflow for the NN configurations used in our evaluation (see Table III).

A PE generates requests for more input–weight pairs as needed. When finished, the inner product is passed to a PE-local activation function unit that performs a 7-part piecewise linear approximation of a sigmoid (output values are $[0, 1]$) or a symmetric sigmoid (output values are $[-1, 1]$). Piecewise linear parameters are stored in PE-local look-up-tables (LUTs). Consequently, new activation functions can be supported with additional LUT entries. The steepness of the activation function (set when the NN is allocated) is limited to powers of 2 between $1/16$ and 8. When computation finishes and the output has been sent to its destination, the PE transitions to an unallocated state and can be reallocated by DANA's control logic.

G. Control Logic

DANA's control logic handles all tasks related to processing entries in the Transaction Table. This includes choosing which Transaction Table entry to process, generating requests to setup the Configuration Cache, reserving blocks in the Register File, and assigning neurons to PEs. Every clock cycle, the control logic selects a transaction from the Transaction Table using round robin arbitration. Control logic then performs one of the following actions in order of decreasing precedence:

1) **Cache Response** – If the control logic has received a response from the Configuration Cache the control logic updates the associated Transaction Table entry.
2) **Register File Response** – If the Register File responds with an index to a reserved register, the control logic updates the associated transaction.
3) **Cache Check** – If this is a new transaction, the control logic doesn't know if its NN configuration is in the Configuration Cache. The control logic queries the Configuration Cache to find out if it has the specific NN configuration.
4) **Cache Query** – If the selected transaction needs new layer information, the control logic queries the Configuration Cache.
5) **Register File Request** – If a layer needs registers to be reserved, the control logic sends a request to the Register File.
6) **PE Allocation** – The control logic allocates a free PE to the next logical neuron of a chosen transaction.

DANA's control logic repeats these operations so long as there are valid entries in the Transaction Table.

IV. Evaluation of the DANA Architecture

We designed a parameterized, SystemVerilog version of the dominant computational portions of X-FILES/DANA hardware, specifically, a Transaction Table and DANA. We

Table III
NN CONFIGURATIONS FROM BENCHMARK APPLICATIONS USED TO EVALUATE A SUBSET OF POSSIBLE DANA ARCHITECTURES.

Area	Application	Configuration	Size	Description
ASC [24]	`3sum`	$85 \times 16 \times 85$	large	Test if a multiset satisfies 3-subset-sum property
	`collatz`	$40 \times 16 \times 40$	large	Search for counterexamples to the Collatz conjecture
	`ll`	$144 \times 16 \times 144$	large	Compute energies of linked list of Ising spin systems
	`rsa`	$30 \times 30 \times 30$	large	Brute-force prime factorization
Approximate Computing [18]–[20]	`blackscholes`	$6 \times 8 \times 8 \times 1$	small	Financial option pricing
	`fft`	$1 \times 4 \times 4 \times 2$	small	Fast Fourier Transform
	`inversek2j`	$2 \times 8 \times 2$	small	Inverse kinematics
	`jmeint`	$18 \times 16 \times 2$	medium	Triangle intersection detection
	`jpeg`	$64 \times 16 \times 64$	large	JPEG image compression
	`kmeans`	$6 \times 16 \times 16 \times 1$	medium	k-means clustering
	`sobel`	$9 \times 8 \times 1$	small	3×3 Sobel filter
Physics [32]	`edip`	$192 \times 16 \times 1$	large	EDIP approximation of DFT potential energy

excluded other X-FILES hardware components from this evaluation due to their pass-through nature (Transaction Queue and ASID–NNID Table Walker) or their small architectural footprint (supervisor registers). We determined this system's power consumption and maximum operating frequency with a 40nm GlobalFoundries process and Synopsys standard cells using Cadence SOC Encounter. We considered all memory blocks (IO Storage, Cache Storage, and the Register File) as black box SRAMs and computed their power with Cacti [33]. We evaluated the performance of DANA using Modelsim, with clock periods corresponding to the placed-and-routed design, when executing NN configurations taken from application workloads found in the literature (see Table III). We also provide a short comparison with a pure-software implementation of large NNs running on a simulated processor. We primarily focus on the evaluation of the simultaneous multiprocessing components of our architectural contributions (X-FILES arbiter and DANA) to highlight how this architecture aligns with our envisioned trend towards many processes wanting to access NN accelerator resources. Additionally, by designing an architecture with multiprocessing in mind, we can improve the overall throughput of the accelerator.

For our design space exploration, we varied the number of PEs for block sizes of four and eight while measuring power and performance. Each power data point was computed using a separate run of our ASIC toolflow with different DANA parameters. Other architectural parameters were held constant at two Transaction Table entries, four Configuration Cache entries, 32KB of cache storage per entry, and 64 total register elements. Table III shows a list of the NN configurations, taken from recent MLP configurations used in approximate computing [18]–[20], microprocessor state prediction [24], and science [32] applications. Figure 10 shows the average power consumption for each point in the DANA design space. The associated performance, quantified as transaction processing time, is also shown.

Overall, the cache was the dominant contributor to the total power for the design space that we investigated. Power consumption grows approximately linearly as we increase the number of PEs. This linearity, while somewhat unexpected, can be explained due to the architecture. All PEs share a dedicated communication bus where only one PE can generate requests for data to other modules per cycle. Consequently, adding PEs increases the total PE logic linearly, but does not affect any other modules. The power consumed by the Control Logic, Transaction Table, and Register File is less than 10% of the total power due to the small size of their storage arrays and the limited logic footprint of these elements. When moving from four to eight elements per block we see a mean speedup of 28% with an included mean increase in clock period of 2%.

To demonstrate that DANA can efficiently process disparate workloads and that the multiprocessing nature of DANA is sound, we explore the throughput of DANA under different application workloads. We define throughput as the average number of NN edges (input–weight products) computed per cycle for the duration of an application's runtime excluding IO and cache loading overheads. We only count edges towards throughput as these are the only mathematically required operations (less activation functions) of NN computation. We purposefully exclude architecturally-dependent operations (e.g. PE queries of the Configuration Cache) to provide a hard comparison against an ideal NN accelerator not limited by architectural constraints.

Figure 11 shows the throughput when executing single-application workloads from the NNs in Table III. The theoretical maximum throughput is approximately equal to the number of PEs after considering processing overheads and the cost of computing an activation function. Throughput expectedly varies with application and design space variations. A dominant factor is the amount of *independent* work that each PE can execute and how frequently it needs new data. Independent work can be viewed as the number of incident

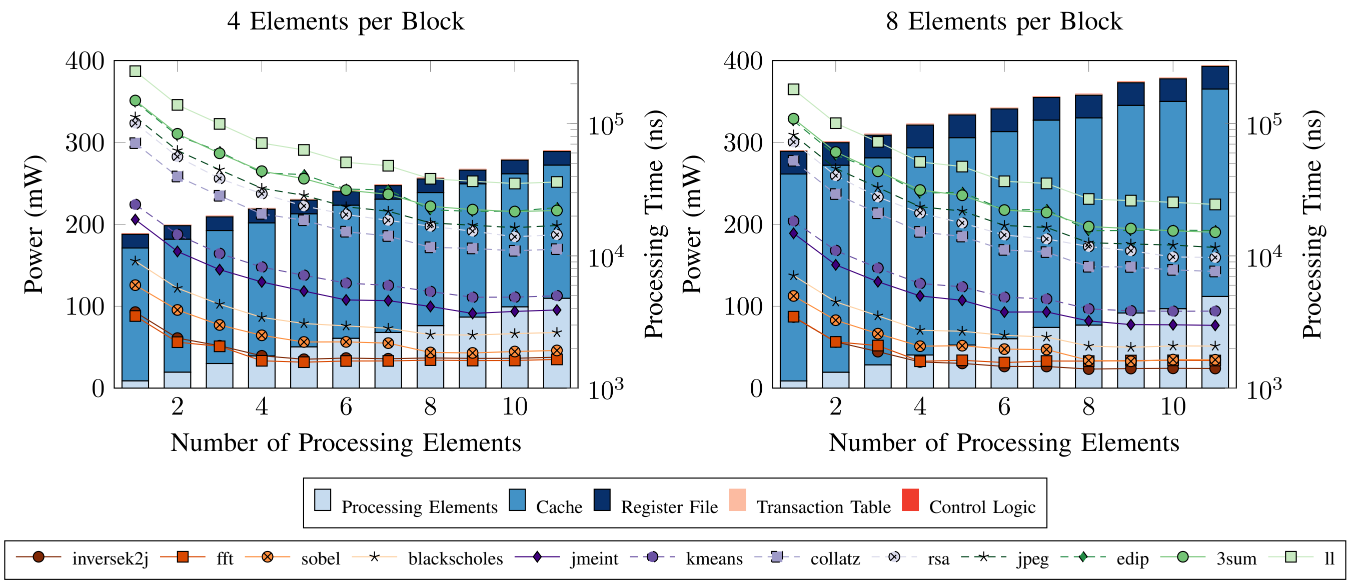

Figure 10. Average power per module (bar plot) and processing time (line plot) of different NN configurations, listed in Table III, when running exclusively (i.e., one NN at a time) on DANA architectures with 1–11 processing elements and four or eight elements per block.

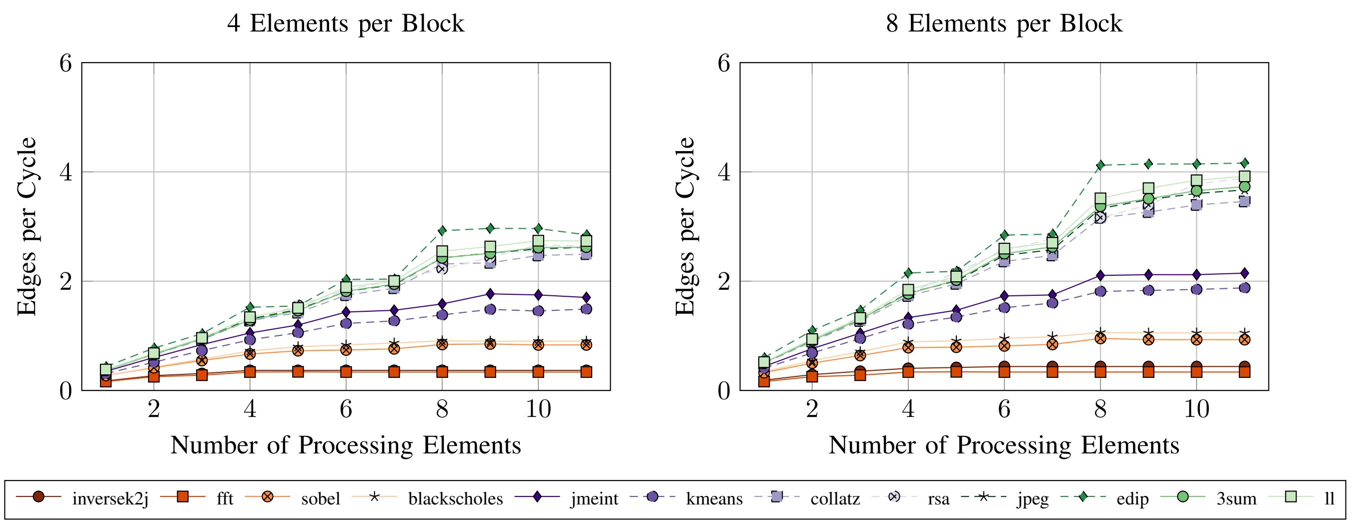

Figure 11. DANA's throughput, measured in edges per cycle, when executing one instance of an NN transaction.

edges to each logical neuron and is a characteristic of NN topology. For example, the wide difference in throughput between `edip` and `inversek2j` is due to the amount of work that each neuron must perform. Only considering the hidden layer, each neuron in `edip` computes 192 input–weight products while each `inversek2j` neuron only performs two. The underlying topology of `inversek2j` causes setup overhead—allocating a PE, having the PE request neuron information from the Configuration Cache—to be of similar duration to the actual computation and hence, drive down throughput.

Increasing the block width can help amortize communication costs (as input or weight data requests will happen less frequently) as evidenced by the increasing performance and throughput of eight elements per block vs four. However, block width does not help small networks like `inversek2j`. Increasing the available bandwidth between the PEs and the Register File/IO Storage and/or the Configuration Cache does not help if it does not fetch more work for a PE. Our current implementation only supports a single PE data access per cycle, but this is not a fundamental limit as all memories are dual-ported. Increasing the effective

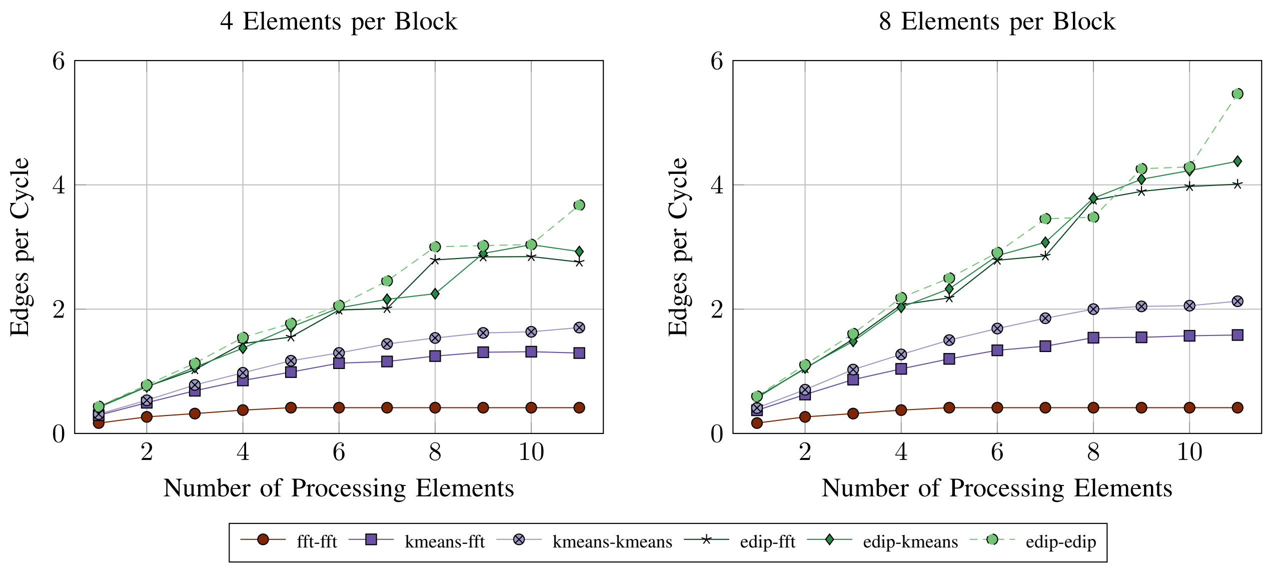

Figure 12. DANA's throughput, measured in edges per cycle, when executing two workloads.

bandwidth beyond two accesses requires data replication, memory overclocking, or grouping like PE requests together.

While we see dramatically different throughput for different applications, we postulate that the inefficiencies of an NN like `inversek2j` can be amortized if running a transaction with more computational work per neuron. To test this, we evaluated DANA when running simultaneous NN processing workloads for representative large (`edip`), medium (`kmeans`), and small (`fft`) NNs. Results are shown in the Figure 12. Running disparate workloads results in a small loss in maximum individual throughput, e.g., running `edip` with `fft` yields a lower throughput than just running `edip` alone. However, and critically, running two instances of the same transaction results in throughput speedups as high as 30% (for two `edip` transactions) and 6% on average. These results are shown in Figure 13.

In certain cases we do see a noticeable throughput decrease, specifically for the case of a DANA instance with eight PEs running `edip-kmeans` with four elements per block or `edip-edip` with eight elements per block. In both of these tests, the first `edip` transaction starts slightly ahead of the other transaction. The control logic's round robin arbitration allocates the output neuron of the first `edip` transaction before all neurons in that transaction's hidden layer have finished. The long running nature of an `edip` hidden neuron causes this output PE to sit idle while its inputs from the hidden layer are computed. This throughput slowdown is exacerbated by the fact that both `edip` and `kmeans` have hidden layer sizes that are multiples of the number of PEs causing them to run efficiently back to back. Restricting PE allocation to only occur after all neurons in the previous layer have finished remedies this problem.

Overall, these results indicate that exposing more than one transaction to DANA allows better utilization of computational resources. Conversely, an NN accelerator without multithreading support must execute separate NN transactions sequentially and cannot assign unused computational resources to other transactions.

Table IV
ENERGY, DELAY, AND EDP REDUCTIONS WHEN EXECUTING NNS ON DANA COMPARED TO A PURELY SOFTWARE IMPLEMENTATION.

NN	Energy	Delay	EDP
`3sum`	7×	95×	664×
`collatz`	8×	106×	826×
`ll`	6×	88×	569×
`rsa`	6×	88×	566×

Finally, we evaluated the power–performance of our DANA architecture compared to the FANN library running on a single core (one Intel SCC core) using the gem5 simulator [34]. The only software optimization used is FANN's built in software pipelining. These results are shown in Table IV. We estimate DANA–memory latency as 100ns plus the time it takes DANA to read data blocks from a clock crossing FIFO at its own clock rate. As expected, our dedicated hardware implementation uses an order of magnitude less energy and is two orders of magnitude faster than a software implementation running on a general-purpose processor.

In summary, these improvements indicate the soundness of the DANA architecture and its ability to enable NN transaction multiprocessing. DANA's throughput can be further improved by increasing internal communication bandwidth, allowing multiple PEs to be allocated per cycle, and improving PE allocation logic.

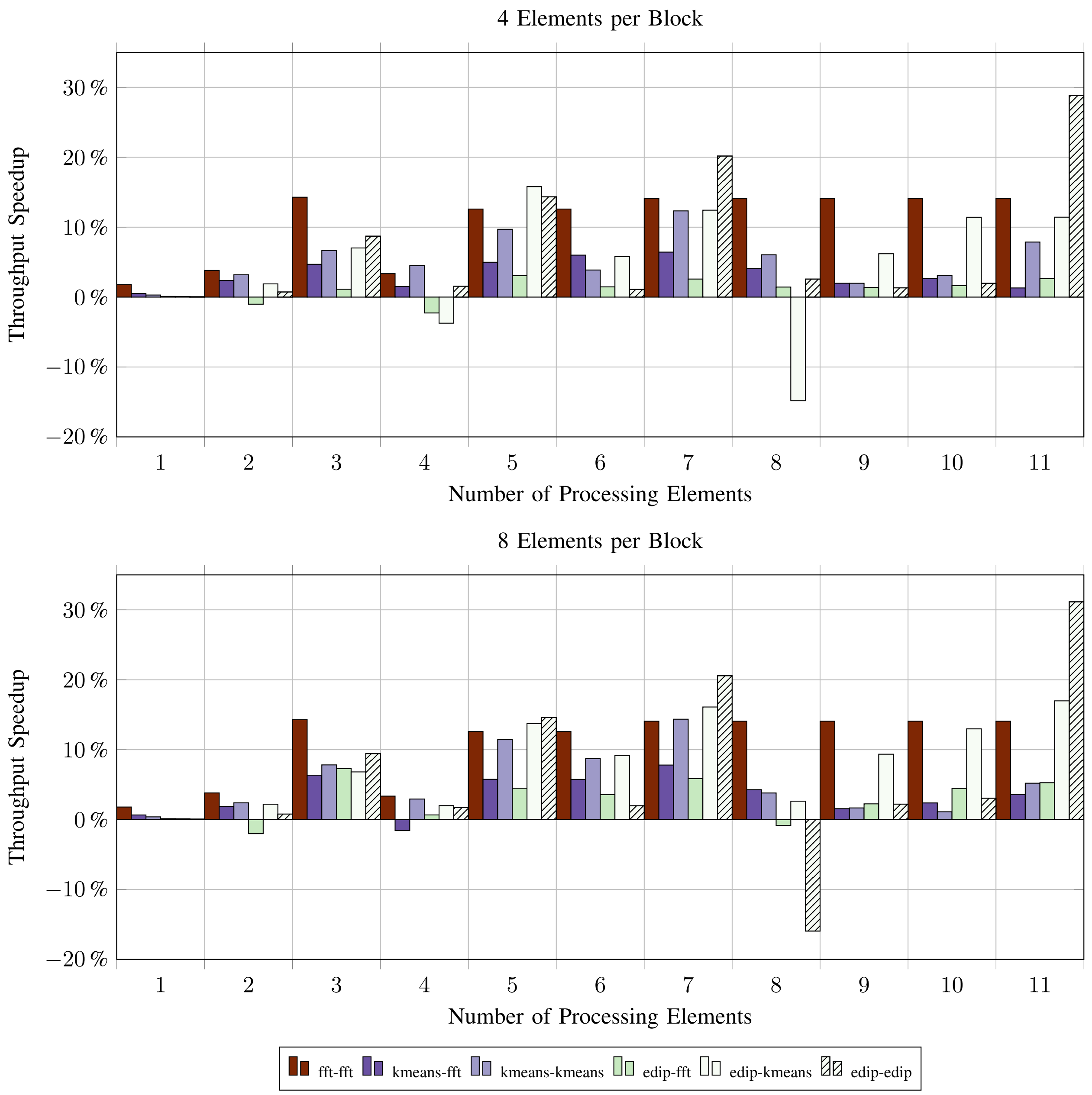

Figure 13. DANA's percentage throughput speedup when running two simultaneous transactions vs. the same transactions serially.

V. Related Work

A. Accelerator Interfaces and Management

The work proposed in this paper does not define a transport layer (like RoCC [25]–[27] or AXI [35]), but a set of extensions that live on top of the transport layer. In this respect, the X-FILES are related to similar extensions or frameworks like HiPPAI [29] and ARC [30]. However, the needs of HiPPAI and ARC are distinctly different from the X-FILES. HiPPAI is strictly focused on reducing the overheads associated with using hardware accelerators while ARC is concerned with the management of a sea of similar or dissimilar accelerators in a multicore system. In our model, we deal with an accelerator which, due to the temporal locality of NN configuration information, has some benefit being shared simultaneously between multiple threads or cores as opposed to being shared in time. Furthermore, the underlying accelerator architecture (as we demonstrate with DANA) can benefit from running in a simultaneous multiprocessing mode that increases the throughput of the underlying resource. Similar to VEAL [36], we provide a way of abstracting away the underlying hardware resources from the world of a software developer. However, we provide a complete description of access models involving register and memory transfers to better enable the library writer or hardware designer to choose the best interface mode on a per-application basis.

B. Neural Network Accelerators

Much work has already been completed on the design of dedicated hardware for acceleration of MLPs [18]–[23] or other types of NNs, be they Convolutional Neural Networks [5]–[9], Deep Belief Networks [10]–[12], Hierarchical Model and X [13], or more biologically accurate models [14]–[17]. We, however, propose what we believe is the first instance of an NN accelerator architecture that supports the simultaneous execution of multiple NNs. While evaluated only on MLP workloads, the generic nature of the architecture (that it processes arbitrary graphs described by an NN configuration) makes it highly extensible to other NN flavors like Convolutional Neural Networks and Deep Belief Networks. We view DANA, and the clear definition of the X-FILES, as an additional step towards wide adoption of NN-like and machine learning computation in everyday workloads through tight integration with modern microprocessor architectures.

VI. Conclusion

We propose the X-FILES, a set of hardware/software extensions for general machine learning computation, and DANA, a new NN accelerator architecture that supports simultaneous multiprocessing of NN transactions. The X-FILES provide a generic way to describe and communicate NN transactions to a shared NN accelerator resource. By nature, the X-FILES expose multiple possible transactions to a backend accelerator. Our new accelerator architecture, DANA, takes advantage of this to simultaneously process multiple NN transactions and improve its overall throughput. While DANA was only evaluated on MLPs, the X-FILES are independent of NN flavor and DANA can be modified to support multiple NN types through planned extensions for Convolutional Neural Networks, Deep Belief Networks, and other NN types. Additionally, extensions to DANA to provide core-assisted or hardware learning are anticipated. Furthermore, other uniprocessing or multiprocessing accelerators aligning with our NN transaction model can be interfaced and used with the X-FILES. We view this work as an additional step, furthering that of others, towards generalizing the way in which NN computation is described, viewed, and managed by software, operating systems, and underlying NN accelerator architectures.

Acknowledgment

We would like to thank Christopher Batten and Scott Beamer for their timely, valuable feedback. We would also like to thank the anonymous reviewers for their helpful comments. 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] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 3642–3649.

[2] Y. Kara, M. A. Boyacioglu, and Ö. K. Baykan, "Predicting direction of stock price index movement using artificial neural networks and support vector machines: The sample of the istanbul stock exchange," *Expert Systems with Applications*, vol. 38, no. 5, pp. 5311–5319, 2011.

[3] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 369–397, November 2002.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle *et al.*, "Using machine learning to focus iterative optimization," in *Proceedings of the International Symposium on Code Generation and Optimization (GCO)*, 2006, pp. 295–305.

[5] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1915–1929, 2013.

[6] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2010, pp. 247–257.

[7] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, May 2010, pp. 257–260.

[8] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Conference on Computer Vision and Pattern Recognition Workshops*, June 2011, pp. 109–116.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 269–284.

[10] S. K. Kim, L. McAfee, P. McMahon, and K. Olukotun, "A highly scalable restricted boltzmann machine fpga implementation," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2009, pp. 367–372.

[11] D. L. Ly and P. Chow, "High-performance reconfigurable hardware architecture for restricted boltzmann machines." *IEEE Transactions on Neural Networks*, vol. 21, no. 11, pp. 1780–1792, Nov 2010.

[12] L.-W. Kim, S. Asaad, and R. Linsker, "A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 1, pp. 5:1–5:23, 2014.

[13] M. S. Park, C. Zhang, M. DeBole, and S. Kestur, "Accelerators for biologically-inspired attention and recognition," in *Proceedings of the Design Automation Conference (DAC)*, 2013, p. 135.

[14] J. Fieres, J. Schemmel, and K. Meier, "Realizing biological spiking network models in a configurable wafer-scale hardware system," in *International Joint Conference on Neural Networks*. IEEE, 2008, pp. 969–976.

[15] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras *et al.*, "Spinnaker: mapping neural networks onto a massively-parallel chip multiprocessor," in *International Joint Conference on Neural Networks*. IEEE, 2008, pp. 2849–2856.

[16] J. Seo, B. Brezzo, Y. Liu, B. Parker, S. Esser, R. Montoye *et al.*, "A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *CICC*, sept. 2011, pp. 1 –4.

[17] S. Kestur, M. S. Park, J. Sabarad, D. Dantara, V. Narayanan, Y. Chen *et al.*, "Emulating mammalian vision on reconfigurable hardware," in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 141–148.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 449–460.

[19] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *Proceeding of the International Symposium on Computer Architecuture (ISCA)*, 2014, pp. 505–516.

[20] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze *et al.*, "Snnap: Approximate computing on programmable socs via neural acceleration," in *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 603–614.

[21] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, "Memristor-based approximated computation," in *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*, 2013.

[22] X. Liu, M. Mao, H. Li, Y. Chen, H. Jiang, J. J. Yang *et al.*, "A heterogeneous computing system with memristor-based neuromorphic accelerators," in *Proceedings of the High Performance Extreme Computing Conference (HPEC)*, 2014.

[23] S. Eldridge, F. Raudies, D. Zou, and A. Joshi, "Neural network-based accelerators for transcendental function approximation," in *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, 2014, pp. 169–174.

[24] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer, "Asc: Automatically scalable computation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 575–590.

[25] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the Design Automation Conference (DAC)*, 2012, pp. 1216–1225.

[26] H. Vo, Y. Lee, A. Waterman, and K. Asanović, "A case for os-friendly hardware accelerators," in *Workshop on the Interaction Between Operating System and Computer Architecture*, 2013.

[27] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "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.

[28] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual volume ii: Privileged architecture version 1.7," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-49, May 2015.

[29] P. M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer *et al.*, "Hippai: High performance portable accelerator interface for socs," in *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2009, pp. 109–118.

[30] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Architecture support for accelerator-rich cmps," in *Proceedings of the Design Automation Conference (DAC)*, 2012, pp. 843–849.

[31] S. Nissen, "Implementation of a fast artificial neural network library (fann)," Department of Computer Science University of Copenhagen (DIKU), Tech. Rep., 2003, http://fann.sf.net.

[32] J. F. Justo, M. Z. Bazant, E. Kaxiras, V. V. Bulatov, and S. Yip, "Interatomic potential for silicon defects and disordered phases," *Physical Review B*, vol. 58, pp. 2539–2550, Aug 1998.

[33] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

[34] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[35] "Amba axi protocol specification," *ARM*, June 2003.

[36] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008, pp. 389–400.