University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Brain-Flow : A brain inspired dataflow implementation using DEMAC

Diego Roa, Ryan Kabrick, Siddhisanket Raskar, Jose M Monsalve Diaz and Guang Gao

CAPSL Technical Memo 134
October, 2019

Contents

Abstract

Inspired by the structural and semantics similarities between Neural Networks and Dataflow Networks (driven by dataflow program execution models), general AI algorithms with human-like intelligence can benefit greatly from Dataflow-inspired multi-grained programming and program execution models (PXMs) and architectures. However, most commercially available commodity hardware architectures currently lack crucial capabilities which would potentially benefit the exploration and experimentation of multi-grained dataflow execution models. For this reason, programming and runtime systems based on dataflow models have to emulate the desired architecture and hardware behavior through software, significantly increasing the overhead in the final application, tainting the validity of evaluations and results cultivated from the aforementioned approaches. We propose an alternative path to solve this problem based on a software-hardware co-design principle.

The goal of this project is to work towards a hardware-software co-design that explores the benefits of the dataflow-inspired fine-grain synchronization schemes found within dataflow PXMs when applied to brain-inspired algorithms and software applications. To this end, we have developed the underlying computational tools by combining FPGA implemented components used by a dataflow inspired runtime system which works to implement the Codelet Model[]. We have then applied it to a traditional computer vision and natural language processing by porting a face landmark estimation algorithm into DARTS[], a Dataflow-inspired simulation framework implementing Codelet Model targeting x86 (64-bit) architectures.

DEMAC[] is an extendible, 3D printed platform that is composed of 24 highly-parallel embedded system boards organized into a Beowulf-cluster. Each node is a commercially available embedded-system board that consists of a dual-core main processor, programmable logic system and a multicore accelerator chip with a theoretical capacity of 2 GFLOPS. The open-source software stack along with the hybrid-architecture of the embedded-system board provides the flexibility to define custom hardware-software mechanisms which rely on Dataflow principles.

1 Introduction

Introduction goes here.

2 Background

In this section, we give a brief overview of the basic terminology required for a contextual understanding of the core principles underlying our project.

2.1 Codelet Model

The Codelet Model [] [] is a hybrid von Neumann/Dataflow Program Execution Model (PXM) that differs from a sequential computer's PXM in the following aspects:

- Instead of a single program counter, there can be multiple program counters, allowing for concurrent execution of instructions from multiple parts of the program
- Programs are divided into small sequences of instructions in a two-level hierarchy: Threaded Procedures (TP) and Codelets.
- The order of execution among codelets is determined by data and control dependencies explicitly identified in the program, as opposed to the program order
- Frames holding local context for functions are allocated from a heap rather than a linear stack

These features provide the operational semantics, with enough detail, so a programmer can accurately predict a given program's behaviour at a given point in the program's execution. It provides a basic set of primitive operations, which give specific details of instructions, registers as well as their interactions to support concurrent execution of Codelets.

2.1.1 Codelets

Based on Semi-Dynamic Dataflow principles, a Codelet is the combination of instructions, data dependencies and the definition of operational conditions. This definition allows for the reduction and the amortization of the synchronization overhead while also improving data locality. Codelets can be executed when all data dependencies have been met and there are enough computational resources available. The Codelet Model maintains the ordering constraints among instructions within one codelet, but loosens the constraints between different codelets. A Codelet is a sequential, non-preemptive, atomically-scheduled set of instructions. Representing a program as multiple sections of code allows the execution of instructions as soon as the data and resources are available.

- **Sequentially Executed:** Modern processors perform sequential execution very efficiently, even when there are many dependencies among the instructions and can take advantage of the data locality which is usually present due to these dependencies. Leveraging instruction level parallelism (ILP) and the control mechanisms present in this kind of hardware provides desirable levels of performance without the need for additional, specialized hardware
- **Non-Preemptive:** Once a Codelet begins execution, it remains active in the CPU until it is finished executing.
- **Atomic Scheduling:** A codelet cannot be interrupted, therefore it should not begin execution until it is guaranteed to finish without interference from other processes

2.1.2 Threaded Procedure (TP)

A TP consists of one or more Codelets. In the Codelet Model, a Codelet is always a part of an enclosing TP. All codelets in a procedure share the local variables and input parameters of that TP, much like context in modern programming languages. Objects may have a lifetime beyond a single procedure, may have a size which can't be determined at the time the frame is allocated, or may be shared among multiple procedures. Some objects may even need to exist outside of a particular frame's context. The tight coupling between codelets within these categories requires a fine-grain level of synchronization and data sharing mechanisms.

TPs and Codelets differ in their contexts, their lifetimes and their manner of invocation. The context of an instance of a TP is similar to the context of a function call in a conventional language such as C. Codelets have much smaller context, consisting only of registers and specialized state variables (SyncSlots). On the other hand, a TP instance remains "live" even if none of its Codelets are active or enabled; a TP must also explicitly terminate itself. TPs are invoked explicitly by the application program.

When the program invokes a TP, the machine creates a context for this procedure, initializing the input parameters with the values passed to this TP. A new frame is allocated in memory for this particular instance of the procedure. All codelets can access this frame through a unique frame identifier (FID), which is part of the invoking Codelet's context and is normally kept in a register for quick access. This is similar to the frame pointer found in conventional block-structured languages. Given the FID, it is possible to access any local variable or input parameter within the corresponding TP's instance context. This also allows the use of recursive calls. Because procedures are explicitly terminated, no garbage collection of frames is needed.

2.1.3 Sync Slots

A codelet needs to be sure that all dependencies have been satisfied before it is enabled, since the Codelet can't be preempted once started. A counter is used to count the incoming signals so it is known when a Codelet is ready to be enabled. The Codelet model allows for the instantiation of SyncSlots to control different Codelets simultaneously along with allowing several counters to control the same Codelet.

A SyncSlot contains:

- **SyncCount (SC):** Indicates the number of dependencies to be received by the SyncSlot before the specified Codelet can be enabled.
- **ResetCount (RC):** If the SyncCount reaches 0 the Codelet specified by the SyncPointer is enabled. After the Codelet execution starts, the SC is set back to the RC.
- **SyncPointer (SP):** Binds the SyncSlot to one of the Codelets in the TP.

The SyncSlot counts the number of tokens received. Sending a SyncSignal is sufficient if and only if both sender and receiver are in the same TP instance. This is because local variables can be used to transfer the data from one Codelet to another. Because the SyncSlots are used for controlling the enabling of Codelets, they must persist beyond the lifetime of a Codelet, and therefore must be part of the context of a TP. The use of a RC allows Codelets to be enabled multiple times.

Each Codelet is part of an enclosing TP, a given instance of a Codelet is associated with one particular procedure instance. Each Codelet is given a unique Codelet identifier (CID). Each instance of a codelet can be uniquely identified by a pair (FID, CID).

2.1.4 Codelet Abstract Machine

What this model has in common with other parallel paradigms is the division of a program into multiple sections of code, defined as Codelets. Combining individual instructions into Codelets reduces and amortizes the overheads of synchronization and improve data locality. A Codelet that is not ready to begin execution is classified as dormant. When the system decides a codelet is ready to execute, it enables the Codelet. Since the CPU may still be busy with other codelets at that time, there may be a delay between the time a codelet is enabled and the time it starts running. We refer to the first state as enabled while the latter state is considered active.

Figure 2 shows an abstract model of a machine for executing Codelets. There are two pools of Codelets: one for dormant Codelets and the other one for enabled codelets. When Codelets are enabled, they are moved from the dormant pool to the enabled pool. When the Active Codelet Processor has available resources for executing the Codelet, it takes one from the enabled pool, making it active. At the same time, it must allocate the required resources needed by this Codelet and begins execution.

A two-layer hierarchy of Codelets and TPs adds flexibility, programmability and support for current off-the-shelf processors. When a procedure is invoked, the system must first allocate and initialize the frame. Once this stage is completed, the initial Codelet is enabled, and begins waiting for the machine to have sufficient resources available for execution. All other Codelets in the procedure must rely on other mechanisms to become enabled.

The machine must check and verify dynamically that all data and control dependencies have been satisfied before enabling a Codelet. Using Synchronization Signals (SyncSig) and Synchronization Slots (SyncSlots). A synchronization signal, or SyncSig, is sent from one Codelet to another, either in the same or another procedural instance, to alert the recipient that a specific control or data dependency has been satisfied.

2.1.5 Memory Model

Most large multiprocessors use either distributed memory or distributed shared memory. Distributed memory architectures have separate memories for each processor or group of processors. These are separate both physically and logically. A process can access data in remote memory only indirectly, by communicating with a process that has access to that memory (often referred as message passing machines). Distributed shared memory architectures have a global address space, which allows any processor to access any memory location in the system. The Codelet PXM is neither restricted to distributed memory machines nor limited by them.

To support the Codelet PXM, a machine's memory system must have the following properties:

- An active Codelet must have direct, low-latency access (through load and store operations) both to its private Codelet context and to the frame it shares with other Codelets in the same procedural instance
- An active sequential function call must have direct, low-latency access both to its local linear stack and to the frame belonging to the Codelet which initiated the sequential function call (either directly or through other function calls)
- Instruction pointers are uniform through the system. The code for all threaded procedures and sequential functions is accessible from all processing elements of the machine, and given SP value has the same meaning on all such elements. The instruction addresses used in sequential function calls must be the same on all processing elements.
- All objects in the system which may be accessible by more than one TP. This includes FID, SyncSlots or any data addresses which may be bound with SyncSignals; though they must be globally unique and accessible by special Codelet Model operations.

Since all Codelets within a TP share data in a single frame, all Codelets in a given TP must run on a processor or processors with access to the same context. The first three requirements ensure rapid execution within a Codelet and guarantees that procedures and sequential functions can be invoked on any processing element. This allows for Codelets to communicate seamlessly with other procedures. Each processor must be able to determine the exact location of any given memory reference.

2.1.6 Communication Mechanisms

Each element (TP, Codelet, SyncSlot) and each instance of those in the PXM is referenced by a unique identifier. Memory references, at least those passed between procedures, must be globally unique across the machine, and each processor must be able to determine the exact location of any given memory reference. The Codelet Model Defines atomic operations for sending data and a SyncSignal together. This Data-Transfer/SyncSignal operation may be initiated by the producer of the data, which sends local data to another location, or by the consumer, which sends a request for remote data to the system. The latter operation is called a split-phase transaction because the request and data transfer may occur in distinct phases.

2.1.7 Operations

The execution of a Codelet-based program relies on various operations for sequencing and manipulating the Codelets in this PXM. These operations perform the following functions:

- Invocation and termination of TPs and Codelets
- Creation and manipulation of SyncSlots
- Sending of SyncSignals to SyncSlots, either alone or atomically bound with data.

2.2 The Embedded System

The embedded system board used is an affordable, highly parallel and open source embedded system. The board features a programmable logic/main processor combined chip, a multi-core accelerator, and some peripherals. The accelerator operates on a RISC architecture and is used to offload computation for acceleration. It is the accelerator, together with its low cost and the openness of its software stack, that makes the embedded system unique and a great fit for our purpose. The board runs full-fledged Ubuntu 15.04, modified to contain the drivers that communicate with the accelerator through some programmable logic running on the FPGA.

Each of the cores of the accelerator has 32 KB of scratchpad memory that can be accessed by other cores. There is no coherency protocol, the memory model is relaxed and there are a couple of instructions that allow atomic access. The user is responsible for managing this memory and use it for code and data. For programming the system it is possible to use C, some limited functionalities of C++, and assembly. However, all of the software stack (i.e. operating system, drivers, libraries, and FPGA code) is fully open source.

Thanks to these properties, the system is highly flexible and customizable, while its bare-bones design allows us to remove some of the burden present in current microprocessors (e.g. cache coherency protocols) that have shown to be detrimental to the performance of the Codelet Model implementations. However, it also makes the realization of this project more challenging.

2.3 Programming Models

2.3.1 MPI

A standard Message Passing Interface for distributed memory concurrent computers and networks of workstations. Provides the communication and synchronization mechanisms to share control and data among the different nodes through the network. The main advantages of using MPI are the portability, ease-of-use and a clearly defined set of routines.

2.3.2 OpenMP

Open Multi-Processing, is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior. It is used for communication and synchronization between the main processor cores in a single node.

2.4 Machine Learning

2.4.1 Protocol Buffers

Protocol buffers (or protobuffers) offer the programmer an intuitive interface to serialize structured data across different languages and platforms in an intuitive, extensible fashion. The programmer will input how they wish for their messages to be formatted, much like XML. Google Protobuffers highlight they are "smaller, faster and simpler" than their XML equivalents [10,11,12]. In comparison to XML formatting, protocol buffers offer a solution that can be 3-10x smaller, resulting in a performance improvement of 20-100x times when parsing or deserializing. Additionally, the syntax of protocol buffers is less ambiguous than its counterpart XML files. As an example, figures 4 and 5 are both representation of a class Person with fields name, and email in protocol buffers and XML respectively.

A second advantage of Protocol Buffers is that it is meant to create interfaces that natively map into different programming languages. Therefore, the same protobuffer description file can be used to generate native libraries in C, C++, Java, Go language and others. Therefore improving programmability, usability and performance. All of this configuration is carried out in the .proto file for the project. Within this one file, multiple languages are easily and comprehensively generate applicable source code.

Protobuffer for Neural Networks.

Protocol buffers are a general method to describe any sort of information. The programmer must first specify how they want their data interpreted in terms of messages. However, Google has used protocol buffers for their Tensorflow runtime in order to describe neural networks that are to be executed in the underlying Tensorflow runtime. The magic resides in that protocol buffers support C/C++/Objective-C, Python, Go, Ruby and C. Therefore, it is possible to create front ends in these programming languages that will generate neural networks represented using the graph definitions that can be interpreted by the runtime located in the back end. In the meantime, the back end will implement each of the operations needed for the execution of the neural networks.

Tensorflow's neural networks definitions include the concept of Graph as a collection of nodes. Each node has at least: a list of predecessor nodes, an operation type, and a set of attributes containing additional information of the network. It is possible to store untrained neural networks that contain no particular weights or values representing just the structure of the network. Additionally, it is possible to store so-called “frozen” neural networks that are fully trained model ready to be used for execution.

2.4.2 Facial Landmark Detection Algorithm

Face detection is a problem in computer vision for locating and localizing one or more faces in a photograph. Locating a face in a photograph refers to finding the coordinates of the face in the image, whereas localization refers to demarcating the extent of the face, often via a bounding box around the face. Detecting faces in a photograph is easily solved by humans, although has historically been challenging for computers given the dynamic nature of faces.

For facial landmark detection, we have selected the Multi-task Convolutional Neural Network (MTCNN) algorithm [4] developed by Kaipeng Zhang et al. We chose this algorithm as the neural network is large enough and fast enough exploit advantages of the dataflow based execution model. The MTCNN model consists of 3 separate networks: the P-Net, the R-Net, and the O-Net. Figure 6 briefly shows the architecture of entire neural network.

In the P-Net, for each scaled image, a 12x12 kernel runs through the image, searching for a face. Within each of these 12x12 kernels, 3 convolutions are run through with 3x3 kernels. After every convolution layer, a prelu layer is implemented. In addition, a maxpool layer is put in after the first prelu layer. After the third convolution layer, the network splits into two layers. The activations from the third layer are passed to two separate convolution layers, and a softmax layer after one of those convolution layers. One of the convolution layers outputs the probability of a face being in each bounding box, while the other convolution outputs the coordinates of the bounding boxes.

R-Net has a similar structure, but with even more layers. It takes the P-Net bounding boxes as its inputs, and refines its coordinates. he end, giving out two outputs: the coordinates of the new bounding boxes and the machine’s confidence in each bounding box. Finally, O-Net takes the R-Net bounding boxes as inputs and marks down the coordinates of facial landmarks.

2.5 Dataflow inspired Neural Network chips

Deep learning has emerged as one of the most important computational workloads in recent years. To meet its growing computations demands, we have seen a rise in the development of dataflow inspired chips specialized for processing neural networks.

Cerebras[] is one of the projects where its cores are designed specifically for the sparse linear algebra of neural networks. To take advantage of this sparsity, the core has built-in, fine-grained dataflow scheduling, so that the compute is triggered by the data. We should also note the importance of software stack and Program Execution Model (PXM) highlighted by this project. The Cerebras software stack provides a seamless interface to existing high-level ML frameworks, such as TensorFlow[] and PyTorch[]. The graph compiler begins by extracting a dataflow graph representation of the neural network from the user-provided framework representation.

The Tianjic chip[] adopts a many-core architecture, reconfigurable building blocks and a streamlined dataflow with hybrid coding schemes, and can not only accommodate computer-science-based machine-learning algorithms, but also easily implement brain-inspired circuits and several coding schemes. By forming a parallel on-chip memory hierarchy and organizing the dataflow in a streaming fashion, the Tianjic chip can provide improved throughput and power efficiency.

3 Innovation

The long term objective of this project is to create a distributed embedded systems platform that allows execution of neural networks using the Codelet Program Execution Model. The system takes a neural network defined in terms of Google's protocol buffer description files similar to those used by Tensorflow. However, instead of using the Tensorflow runtime, these files are translated into an extended Codelet Model's API that implements the operations needed by these neural networks. To this end the contributions of this project are:

- Creation of a transpiler that takes frozen neural network models described in Protocol Buffer format and produces a version of this neural network into the Codelet Model's API.
- Design and implementation of a beowulf cluster on a 3D printed, stackable, and expandable frame
- Extension of the already existing DARTS (i.e. an implementation of the Codelet Model) to run on distributed mode and in the embedded system, and its accelerator (coprocessor)
- Using an already existing implementation of the Facial Landmark Detection algorithm for comparison against Tensorflow when running on the main processor architecture

While progress has been made in each of the elements of Figure 7, we are still working towards connecting these elements together towards a holistic solution. This document mainly reflects the solution and design of our system. However, our team is still working towards delivering tangible results in our final presentation.

3.1 The importance of Program Execution Models (PXM)

A program execution model provides a common ground between the programmer and the way a program is executed in a specific architecture. It defines a level of abstraction that allows the user (compiler or programmer) to correctly map computational tasks in a given hardware.

3.1.1 Using dataflow for brain inspired applications

Artificial neural networks (ANN) are computing systems that are inspired by, but not identical to, biological neural networks that constitute animal brains. ANN consists of neurons, connections and hierarchical architectures inspired by neuron systems found in human brains. The resemblance of the structure of Neural network algorithms to the way dataflow programs are described allows exploiting the way tasks are distributed. In principle, dataflow programming models execute operations as soon as the data and resources are available, freeing the program from a sequential execution order. This dynamic nature of dataflow makes the ideal execution model to execute neural networks as it avoids the need to explicitly and statically map these graphs on the modern mesh chips.

3.1.2 The DEMAC cluster

The Delaware Modular Assembly Cluster (DEMAC) is an array of embedded systems that combines the many-core accelerator chip and the embedded FPGA with the flexibility of a complete open source stack. The mount is house made 3D-printed frames allowing low cost implementation and scalability. It is designed to fit 4Us of a standard size rack. The multiple nodes allows us to explore a distributed version of the Codelet Model where there is no notion of shared memory among nodes and the execution of tasks in the multicore-shared-memory embedded system chips. Current implementation allows us to use OpenMP and MPI on the cluster.

The communication between nodes is established using an ethernet switch. A Network File System (NFS) based on sshfs allows all the nodes to access the same executable program.

3.1.3 Construction and assembly

The current version of DEMAC holds 24 embedded systems, each of these is mounted in a Board Tray (BoT) that is labeled with the name of the node (NOPA). A Board Casing (BoC) provides housing for 4 boards, a total of 6 of these frames is used, and they allow the connection of 2 independent Fan Casings (FaC) and 1 Power Casing (PwC). The FaC includes an air input and output that enables for the air to flow and separately cool down a single BoC. The PwC holds a USB hub that provides enough energy to the boards.

The cluster is mounted on top of a switch that provides an ethernet connection between the nodes and the HeadNode. Each board is connected using RJ-45 cables to the switch. The HeadNode allows the interfacing of the users with the other nodes. The power for the boards is independent from the auxiliary systems (Cooling and Network).

3.1.4 eDARTS

One of the major limitations that the accelerator chip is its limited support for C++ features. In addition to this, its memory is flexible, but requires the user or runtime to manage it. Previous implementations of DARTS provided an API that heavily borrowed from inheritance in C++ classes. However, because of its unique architecture it is necessary to recreate the fundamental instruments in DARTS specifically for the accelerator. As a result, we have created eDARTS, a C implementation of DARTS specifically catered to the accelerator chip and its memory architecture. Several various assets have been created to bridge the gap between the architectures including, but not limited to:

- **Queues:** Manages codelets as well as threaded-procedures
- Mutex: Ensure resources are only accessed and modified by one process at a time to maintain data integrity
- **Barrier:** Used to confirm each active process has reached a certain end-point within the program's execution
- **Malloc:** Allows for the management of the scratchpad memory on-board each core of the accelerator (ie. local memory, distributed-local memory, and off-chip: global memory stored in the DRAM)
- **Computational Unit:** Used for designating a given core to be a core for carrying out computation
- **Scheduling Unit:** Iterates through list of codelets, waiting for event signals. For every event signal, it decrements the respective codelet's dependency counter. Once an iterated node has a dependency count of zero, it facilitates the execution then repeats the process until the end of the program

3.1.5 Distributed Execution Codelet Accelerated Runtime for Dataflow (DECARD)

Based on the Codelet PXM, DECARD allows the task scheduling and communication of a multi-node system. The current implementation in DEMAC features two elements with the following functions:

- Node Manager: Running in one of the main processor cores, it interfaces with the accelerator. A TP Queue that act as a FIFO buffer, holds the Threaded Procedures generated by eDARTS that should be executed in other nodes and incoming TPs from other nodes.
- Node Communicator: It is in charge of the communications between nodes (TPs, Sync-Slots and Data) and allocating them in local memory.

The Node Manager relies on OpenMP to share information with the Node Communicator, and make use of the libraries provided by adapteva to manage and schedule tasks to the accelerator. The Node Communicator uses MPI functions to send and receive information to/from other nodes in the system.

4 Solution Methodology

4.1 Challenge Description

Machine learning algorithms build a mathematical models based on sample data, often relying on statistical models and heuristics that computer systems use to perform a specific task without using explicit instructions. The main objective is to generalize from experience the ability of a learning machine to perform accurately on new, unseen examples/tasks after having experienced a learning data set. The learner has to build a general model that enables it to produce sufficiently accurate predictions in new cases.

An Artificial Neural Network is an interconnected group of nodes, called neurons. Each connection can transmit information or signals from one artificial neuron to another, which neurons can process and signal additional neurons connected to it. These models usually require large magnitudes of data and computational power to fit the parameters in most of the neurons. Training is often perform in powerful multi-core computers that can iterate through the training dataset and generate a lighter model. Such a model can be used for posterior inference in less powerful machines.

- Commercially available General-Purpose GPU (GPGPU) allow the use of powerful multi-node architectures in desktop computers and clusters; this enables extensible development and implementation of highly parallel algorithms involving various disciplines. Nevertheless, this hardware is based on SIMD (Single Instruction Multiple Data), limiting the flexibility and scalability for complex non-uniform problems.
- The accelerator provides a multi-core MIMD (Multiple instructions Multiple Data) architecture, enabling increased levels of flexibility, scalability and programmability. This allows for programmers to implement novel runtime adaptations.
- The lack of a hardware-enforced Memory Model (MM) in the accelerator chip permits the implementation of a strong but consistent MM based on dataflow principles. Data coherence is enforced by the underlying mechanisms of the Codelet Model.
- DARTS is a software implementation of the Codelet Model, designed to emulate core Dataflow principles within a single-node-multicore architecture.
- Porting DARTS to the accelerator takes advantage of the multicore architecture and the lack of an enforced memory model to test the efficacy of a dataflow-inspired implementation with the intent to increase parallelism.

- To extend the Codelet Model beyond a single node, a distributed implementation is required. Using the same principles described by the Codelet PXM, we developed DECARD, a software based runtime that enacts simple Dataflow mechanisms.

- Converting Protobuffer files to the Codelet Model allows for a comprehensive approach to mapping previously defined Neural Networks to Dataflow-compatible architectures.

4.2 Solution

Using off-the-shelf hardware to emulate a dataflow machine, as described above, allows the implementation and viability testing of Dataflow-inspired PXM. This method of prototyping enables developers to analyze the efficacy of such a machine while alleviating the burden of manufacturing specialized hardware.

4.2.1 Codelets as neurons

Given the data-driven and event-driven nature of data-flow inspired execution models, it is possible to raise similarities between Codelets and Neural Networks. Three characteristics that are directly comparable are as follows. First, the execution of both Codelets and Neurons only depend on their inputs. There are no side effects on the execution of a Codelet other than its outputs. Second, the activation function of a neuron is comparable with the operation that is described inside a Codelet. Third, the Direct Acyclic Graph formed by a codelet graph is flexible enough to be able to represent a neural network without requiring any modification.

However, there are other aspects that need extra consideration. Information in a neural network is stored in a distributed way. In the codelet model, this distribution could be achieved by using Threaded Procedures, which stores the tokens and necessary context for the execution of Codelets. Additionally, weights connecting neural networks have no direct mapping to Codelets. However, it is possible to either include the weight as part of the Codelet's operation, or extend the concept of Synchronization Slot to account for weights. Furthermore, some Neural Networks might require saving execution state from previous executions (e.g. Recurrent Neural Networks). For these cases it is possible to use Threaded Procedure's storage as storage containing a "Global" state of the program. If information needs to be shared across multiple threaded procedures, it is possible to use the hierarchy formed by threaded procedures to share state across larger groups of neurons (or layers).

Nevertheless, while the structure of both are similar, it is the mapping of the execution of Codelets to the machine that makes the difference. Neural networks describe the operations and their connections of a particular problem. However, Codelets, in addition to describing the computation, provide a well defined execution model that maps this computation into a parallel system based on a given abstracted machine.

4.2.2 Protocol Buffers to DARTS

In the background section regarding Protocol Buffers, various languages are highlighted which can utilize the efficiency of protobuffer files. For any C-Family language, the protoc compiler can be used to generate applicable source files which can be used by neural-network (NN) runtimes (mainly Tensorflow). Though useful, our project aims to eliminate other NN runtimes and instead use an implementation of the Codelet Model [7] [8], a derivative of dataflow; namely, we are targeting translation from a protobuffer binary file to DARTS (Delaware Abstract Runtime System). This requires taking in an NN defined by a pb binary file and porting the data so that it is separated into a combination of codelets and thread-procedures. Using the header files already present within the DARTS library allows for easy templating of a transpiler for converting from protobuffer to DARTS.

DARTS utilizes two main concepts when implementing a dataflow-inspired execution model:

- Threaded Procedure
  - Reset Codelet Dependencies
  - Decrement Codelet Dependencies
- Codelet
  - Status (Enabled, Dormant, Executing)
  - Sync Slot (Dependency Counter)
  - Fire Function (When all necessary dependencies are satisfied, execute Codelet)
  - TP (Threaded Procedure which houses this codelet)

Python's Tensorflow library allows for easy parsing of information within a protobuffer binary file. The programmer is able to see a comprehensive list of all nodes in the network, their inputs, attributes, etc. For the sake of porting to the DARTS runtime, the programmer has all necessary information for generating a codelet, aside from a TP and the output destinations of a given node. The lack of specific TPs is remedied by the programmer having an understanding of the NN algorithm used to achieve the desired result. For example, the facial recognition algorithm, MTCNN, utilizes three different networks (Pnet, Rnet, and Onet). It is natural to divide work into three TPs, though the transpiler can be easily extended to perform a more in-depth analysis which can design the most efficient separation of codelets into TPs to achieve maximum parallelism. The issue of forward dependencies is addressed through a network-wide cross-reference of a node's name with all other nodes' dependencies. This ensures a previous node within the graph knows where to send its output signal to the decrement the preceding node's dependency counter.

4.2.3 DECARD running on DEMAC

DECARD is the adaptation of the Codelet Abstract Machine for the DEMAC cluster. In the innovation section above, the hardware-software codesign is described, as well as the underlying mechanisms which enforce compliance with the Codelet PXM. A set of predefined functions and routines coordinates the successful execution of parallel programs mapped using the primitives provided.

4.3 Results

Running the implemented functions on DEMAC allowed us to test the correct execution of the previously described algorithms. Our long term goal is to test the viability of Dataflow-based machines. At this point we have achieved several independent advances among the different layers. We have been able to:

- Successfully run eDARTS on the accelerator chip, setting up different configurations of Scheduling Units and Computational Units as well as the execution of operations required for a NN algorithm such as Matrix Multiplication, Convolution, Relu, Vector addition, among others.
- Successfully modify the FPGA bitstream to support hardware implementations native to the Codelet Abstract Machine like TPQueues and SyncSlots. Now moving forward to implement Scheduling and Communication modules.
- Successfully run DECARD to establish multi node communication across the cluster, enforcing Dataflow principles and natives, scheduling tasks and keeping track of the tasks spawned from node to node as well as data dependencies.
- Successfully analyzing protobuffers to automatically extract the information required to map NN to the Codelet Model, converting protobuffers messages to corresponding TPs, Codelets and SyncSlots.
- Successfully run an implementation of the face detection algorithm required to extract the information required for the face identification algorithm.

5 Conclusion

Our groups progress thus far shows promising results for complex computation involving operations such as convolutions along with other simultaneous simple arithmetic operations which could benefit greatly from a multi-core accelerator. Though lacking in the ability to carry out complex operations on a hardware level when compared to modern, serial hardware, many brain-inspired applications rely solely on countless simplistic operations which can be easily handled by even the most elementary hardware. This fact is what motivates our group to pursue alternate routes for accomplishing the goals of complex neural networks. Utilizing our transpiler we are able to convert any protobuffer file to the C++ language, enabling the use of alternate hardware (ie. the accelerator architecture) to provide another, more efficient execution of a given program. The performance gain associated with parallelizing remedial computation, especially when applied to matrix operations, will supercede the performance of modern serial processors.

There are many approaches to distributing the computation; these can consist of handling threaded-procedures solely in the main processor while sending codelets through the accelerator; sending threaded-procedures to a given core; or perhaps setting half of the sixteen-cores to be scheduling units and the other to be computational units, assigning one main processor core to be the scheduling unit which distributes the computation to the communication core. It can then delegate computation across the cluster. This would enable what is perhaps the most integrated and efficient way of handling any parallel computation.

6 Outlook

6.1 Technical Outlook

When designing any program or hardware model for parallel computing, the three main analytical approaches a developer can target when creating something new: scalability, portability, and programmability. Our group seeks to address all three of these key characteristics. Our implementation uses OpenMPI, allowing for an extensible interface which can utilize any amount of nodes to distribute computation across. Aside from scalability, our proposed model is also extremely portable. Because of the protobuffer file, our model can be easily extended to the most common programming languages including Java, the entirety of the C-family, along with others such as C for Windows, Python, and Ruby. To this end, almost any experienced programmer will be able to utilize the Codelet Model and all of its theoretical and implementation benefits.

6.2 Application Outlook

Our future work will integrate our current progress with our theories on the Codelet Model. We will create a transpiler which is universal to any protobuffer file which can port to C++ in such a way that is easy for future parallel hardware to carry out. This includes, but is not limited to, coprocessors such as GPUs, dataflow-inspired chips, as well as any future chips with a distributed-memory model. The need for a consistent and reliable program-execution model is growing as more and more companies begin to explore the development of parallel architectures and the Codelet model is one of the most promising avenues. Our project will prove the efficacy of such a model, showing even on serial hardware, similar, if not better performance can be achieved when utilizing a simple parallel architecture such as the multi-core accelerator chip.

7 Team Introduction

The team consists of three members mentored by a senior graduate student.

- **Ryan Kabrick** is a senior undergraduate student at the University of Delaware. He is pursuing a double major in both Computer Engineering and Applied Mathematics. His areas of interest surround brain-inspired computation as well as brain-computer interfaces. He is excited to begin his PhD work in computer engineering following his graduation in May, 2020.
- **Dawson Fox** is a senior undergraduate student at the University of Delaware. He is studying Computer Engineering and is particularly interested in FPGA development and low-level coding. He is excited to apply his skills to high performance computing and interesting new programmable logic applications.
- **Diego Andres Roa Perdomo** is a junior graduate student in the Electrical and Computer Engineering department at the University of Delaware. His areas of interest involve multiple facets of parallel computing, with a special focus on dataflow hardware-software codesign and dataflow based program execution models. He was in charge of the design, modeling, 3D-printing and construction of DEMAC. He is also highly motivated to develop novel dataflow-based tools for artificial intelligence implementations.
- **Siddhisanket Raskar** is a senior graduate student in Electrical and Computer Engineering department at the University of Delaware. His areas of interest are distributed computing, high performance computing and computer architecture, especially dataflow based program execution models. He has previous experience of participation in this competition.
- **Jose Monsalve Diaz** is a final year, senior PhD student in Electrical and Computer Engineering department at the University of Delaware. His areas of interest are multiple areas of parallel computing, with a special focus on dataflow computer architecture design and dataflow based program execution models. He is also a former participant of this competition, previously exhibiting the importance of dataflow based execution models for machine learning applications. Jose has additional experience in multiple parallel programming models such as OpenMP and MPI, with an emphasis in GPU programming.

Acknowledgment

Firstly, our sincerest gratitude and appreciation to Professor Gao, without whose guidance, support, and wisdom, we never could have made such progress. His leadership is genuinely invaluable.

References