DEMAC and CODIR: A whole stack solution for a HW/SW co-design using an MLIR Codelet Model Dialect

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

CAPSL Technical Memo 136 - DRAFT 1
May, 2020

Contents

List of Figures

Abstract

The introduction of the Instruction Set Architecture (ISA) marked an important begining for current computer systems. Thanks to the separation between hardware and software that provides the ISA, there has been tremendous research conducted on novel hardware architectures and new high level software frameworks. With innovation came an ever growing gap between a programs description and how its executed on hardware.

High performance computing, in their desire to reach extreme computational limits, has pushed computer systems to new limits. Scientists running applications on high performance computers expect these systems to deliver a high throughput efficiently and reliably, while maintaining flexibility, programmability and energy efficiency. HPC machines exhibit different features and behaviors based on their architectural characteristics: they could have accelerators or be a collection of computational units with a non-uniform memory access. However, prototyping and developing strategies for HPC systems can be costly and difficult to achieve. Furthermore, the direction of parallel computing, and its importance in HPC, lacks of a unifying and widely acceptable contract (akin ISAs) which distinguishes the program description from the low level execution on target hardware; namely, a program execution model (PXM). By decoupling the description of the hardware infraestructure, the programming API used to program this hardware, and the rules that determine the interaction of a program through execution on this hardware, we could improve programmability, performance and portability. Furthermore distinguish the software development from hardware development.

Aiming to develop a low-cost, flexible and versatile parallel machine for hardware-software co-design of program execution models research, we present DEMAC (Delaware Modular Assembly Cluster) and CODIR (Codelet Intermediate Representation). DEMAC provides the necessary tools to develop and test novel features for the next generation of supercomputers. It includes a set of 3D-printed frames that can house several card-sized multi-core embedded systems. Along with a full open source stack, these small computers feature Parallella boards containing a Zynq-7000 series SoC with a dual-core ARM processor and an FPGA that interconnects with a 16-core co-processor called Epiphany. All individual nodes are connected through an Ethernet network. An initial version of a runtime based on Dataflow's Codelet Model is also proposed. This project aims to provide a flexible research platform that could benefit different research labs looking for low-cost HPC platforms.

CODIR on the other hand is the Codelet Model domain-specific language and intermediate representation built using the already existing Multi-Level Intermediate Representation (MLIR) compiler infrastructure. It aims to progressively lower a high-level language to an intermediate representation well suited for maximizing optimization opportunities available to the compiler while additionally providing a plethora of hardware support. Underlying CODIR is the Codelet Model which we believe to be the most sound program execution model for mapping computation to the machine and utilizing all available hardware resources.

1 Introduction

1.1 CODIR

The arrival of parallelism and heterogeneity to commodity hardware has resulted in the need to adapt software infrastructures to allow for exploitation of the newly available resources. The need to conserve already existing programming models and software infrastructure (i.e. compilers, libraries and runtimes) has hindered progress. The noticeable differences in execution model between Von Neumann-based sequential computation and current parallel architectures, have forced the software ecosystem to provide solutions that bridge the gap between the two. We believe that the lack of a general purpose abstraction that describes the structure of the system as viewed from the perspective of program execution, as well as the behavior of the program during execution, has resulted in a myriad of ad-hoc software implemented solutions which impede wide-spread application to various hardware and performance. This abstraction, which we refer to as the *Program Execution Model*, has a similar role of the ISA in sequential architectures; ultimately allowing hardware and software to be independently developed. However, current parallel execution models often rely on threading [2] and software runtimes implemented as libraries that are linked during compilation. This does not remedy our problem as there still must exist a translation to the sequential execution model. Such approaches limit the ability to implement these execution models through a hardware/software co-design approach.

Machine learning has emerged as one of the most important computational workloads in recent years. To meet its growing computational demands, we have seen a rise in the development of dataflow-inspired chips specialized for processing neural networks. Cerebras [3] is a project 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-grain dataflow scheduling, so that the computation is triggered by the presence of data. The Tianjic chip [4] adopts a many-core architecture, re-configurable building blocks, and a streamlined dataflow execution with hybrid coding schemes. It will accommodate and facilitate both computer-science-based machine-learning algorithms along with several other coding schemes. However, the development of this field still relies on projects that are independently developed. This is natural but is not conducive to scaling of the software infrastructure, it exacerbates the problem of diversity of execution models, and it can potentially impede code portability for future architecture.

In addition to these enormous leaps in ML hardware development, there have also been numerous software frameworks developed such as Tensorflow [5], PyTorch [6], and MXnet [7], as well as several different representations of the neural network graph. These frameworks have designed interfaces between the description of the *Neural Net* (NN) and its execution, allowing creation of different runtime implementations. For example, the Cerebras software stack provides a seamless interface to existing high-level machine learning 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.

Brain inspired computation represents an excellent example of the aforementioned trend. Qu et al. [8] shows the synergistic progress between the history of parallel computing, Machine Learning (ML), and Artificial Intelligence (AI). The recent progress in parallel architectures has been fundamental to the progress of AI. Although, this field has not been immune to the effects surrounding the lack of a common program execution model. The consequences to this problem have been addressed partially by design of abstractions which separate program description to program execution. However, when implementing the runtime through software approaches even for execution on heterogenous hardware, the ad-hoc execution models (e.g. OpenMP, MPI, Intel TBB and Cuda) are still used. Making the ML framework accountable for potential execution model collisions along with any other possible problems which may be encountered is crucial. Furthermore, in the case of AI hardware accelerators where entire NNs are offloaded, the lack of a general purpose abstraction poses challenges to the interoperability with other computing units.

There exists a need for an infrastructure capable of bridging the gap between high level programming languages and frameworks with the low level parallel hardware. The initial effort taken by ML frameworks created descriptions of NN graphs such as those used by Tensorflow in protobuffer format. However, these do not represent a formal intermediate representation description that allows for a compilation framework to be built on top leveraging potential optimizations. Therefore, an opportunity to further analyze the graph at different level representations provides the possibilities of optimizations for domain-specific languages and hardware. This process is known as progressive-lowering and is the crux of the problem solved by Google's Multi-Level Intermediate Representation (MLIR) compiler infrastructure [9]. MLIR provides a standardization of the building blocks required for machine learning applications, to promote scaling and framework independence. While MLIR solves the issues surrounding the creation of domain-specific languages (DSLs) and progressive lowering, a second necessity is the program execution model underlying the execution of a neural network. Consideration must be taken with regards to how the program description is interpreted and executed on the targeted hardware.

Our proposed solution aims to integrate the view of a system from the perspective of the Codelet Program Execution Model (PXM) [10], a hybrid dataflow Von Neumann model, into the MLIR infrastructure to take advantage of progressive-lowering while simultaneously decoupling the PXM programming interface from the runtime. This will allow for exploration of a software implemented runtime, as well as a hardware/software co-design of the PXM. We use Neural Networks as our evaluation framework, taking advantage of the already existing infrastructure for ML in MLIR, together with the natural mapping between neurons and dataflow models of computation.

In Section 2 we will cover the background information necessary to adequately understand the problems we aim to address. Section 3 describes the methodology used when creating the prototype as well as structuring our proposed solution. Finally, in section 8 we will describe current related works and focus on the future plans of the CAPSL group for moving forward with development of our vision for a proposed Codelet Model software infrastructure.

1.2 DEMAC

There is a wide range of applications that benefit from HPC systems, the basic idea of which is to aggregate computer power in a way that delivers much higher performance than a conventional personal computer would. Dissimilar problems can profit from different approaches or architectures. The widespread usage of HPC and the need to push the envelope of computation has boosted the development of heterogeneous architectures, including GPGPUs or FPGAs as accelerators.

Among the different areas involved in HPC, AI usage has been one of the most rapidly growing during the past decade. Several fields like economics, biology and chemistry are using these kind of algorithms to find patterns, perform classification, or simply automate some activities. The similarities in the morphology of neural networks and dataflow graphs have provided motivation to use dataflow inspired runtimes. Neurons leverage some features like data-centric control schemes and fine-grain scheduling and synchronization.

Brain-inspired and neuromorphic chips leverage regular patterns or behaviors in algorithms that are computationally intensive and can be executed by specialized heterogeneous hardware that can increase the throughput of AI applications.

Current off-the-shelf, state-of-the-art hardware profits from features that have been used for generations, providing instruction level parallelism in the processor, multiple levels of memory, and parallelization techniques. Combined with a language and a compiler that can translate to machine language, the user can control the computational resources to some extent. This understanding agreement between the user, the programming language, and how it translates to the operation of the hardware in the target machine is defined as the program execution model (PXM).

A PXM describes computation in a broad way: setting a base with the semantics of the operations that control the machine, the user defines the program to leverage hardware and software features that support its execution. Research towards novel programming models can be diminished by the difficulty and high cost of developing new hardware features, leading to software-only implementations. These do not provide evidence strong enough to support the implementation of new hardware features in consumer-end processors and existent hardware features may hinder this kind of program's execution.

To determine what features improve performance while maintaining programmability, we need to set a test bench with tools to develop and evaluate these features. This platform must support custom low-level Hardware-Software (HW-SW) implementations. Developing hardware at chip level (VLSI) can be costly; an alternative is to use FPGAs to test the operation of hardware mechanisms first. This system must also have a high level of parallelism.

This paper focuses on the implementation of a multi-core platform that allows us to explore a mix of new hardware and software features, based on the Codelet Model and inspired by Dataflow computation, which will be presented in chapter 2 as the background of this paper. Chapter 5.2.1 presents the hardware aspects of the cluster, while Chapter 5.2.2 describes the software. Chapter 5.2.3 describes the abstraction of the machine and runtime as concepts that provide a base for the implementation. Chapter 7 describes results and, finally, chapter **??** concludes with some remarks of our research.

2 Background

2.1 CODIR

In this chapter, we provide essential background necessary for the rest of this paper.

2.1.1 Codelet Model

The Codelet Model [11] [12] [13] is a fine-grain, event-driven Program Execution Model(PXM) designed to fully exploit hardware architectures featuring a high degree of parallelism by decomposing an application into a great number of lightweight asynchronous tasks called *Codelets*. Codelets can be easily scheduled among cores, with dependence on resource availability, proximity to data to compute, and other constraints.

Each Codelet has a collection of data inputs (respectively, outputs) coming from (respectively, going to) other *Codelets*. When both data and resource dependencies are satisfied, a Codelet is said to be *ready* and is scheduled for non-preemptive atomic execution. Also, since Codelets only act on their data inputs and outputs which are local to their execution frame, long-latency memory operations are avoided. Interactions between Codelets are described in the form of a *Codelet Graph* (CDG), a directed graph where nodes represent Codelets and arcs the dependencies between them. The runtime is responsible for the instantiation and scheduling of Codelets upon satisfaction of dependencies.

CDGs live inside *Threaded Procedures* (TPs). A TP is an asynchronous function that acts as a container for a CDG and the data accessed by its Codelets. This provides a second level of parallelism in the Codelet Model that can be exploited for increasing locality of data. The number of TPs within an application, as well as the number of Codelets per TP and statements per Codelet, may vary, providing users with the ability to decompose an application with several degrees of granularity. An important element stored in the TP are synchronization slots, which coordinate the dependencies of each instantiated codelet, and therefore the communication between codelets. Following the argument-fetching model presented by Dennis and Gao [14].

As a PXM, the Codelet Model relies upon an *Abstract Machine Model* (CAM), which is mapped at runtime to the target many-core system. For extended information on the structure of the CAM please refer to [10]. The most important aspect of this abstract machine for this work is the existence of two resources. A Scheduler Unit (SU) and a Computational Unit (CU). SUs are in charge of resource scheduling and management, and CUs are in charge of Codelet computation.

The Delaware Adaptive Runtime System (DARTS) [15] is currently the most faithful runtime implementation of the Codelet Model and its Abstract Machine Model (CAM). It is written in `C++` and distributed as free and open-source software [16]. Through class inheritance, DARTS combines the Codelet Model and Codelet Model Runtime System.

In DARTS, *Threaded Procedures* (TP) and *Codelets* (CD) are defined as class object definitions by the user. Through the use of inheritance and virtual methods, the runtime executes the user defined methods that comprise the firing of a Codelet, as well as the instantiation and resource allocation of a Threaded Procedure. Extensive information about DARTS can be found in [15].

2.1.2 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 decides how they wish for their messages to be formatted, much like XML. Google Protobuffers highlight the fact they are ”smaller, faster and simpler” than their XML equivalents. Another advantage of Protocol Buffers is they permit the creation of 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.

Tensorflow's neural networks provide their own protobuffer description that includes definitions of the Neural Network that includes a 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.

2.1.3 Multi-Level Intermediate Representation Compiler Infrastructure

[9] Many modern compilers, such as Clang, only provide a single abstraction level like LLVM IR. This is an optimal abstraction for low level machine code as it offers numerous optimization opportunities for all machine code generation. It falls short when being the only abstraction layer to higher level languages and frameworks. The larger the gap grows between a programs description and its information about the abstract machine underlying the execution, the less optimized it is for execution on that machine.

It was designed to provide compiler designers access to multiple levels of abstraction to translate and optimize with through the promotion of progressive-lowering, a process of lowering code to different levels of abstraction to achieve domain-specific optimizations. These levels of abstraction are referred to as *dialects*. A dialect in MLIR is a grouping of operations, attributes, and data types under a unique namespace (ie. Tensorflow, TF Lite, LLVM IR, etc.). Dialects

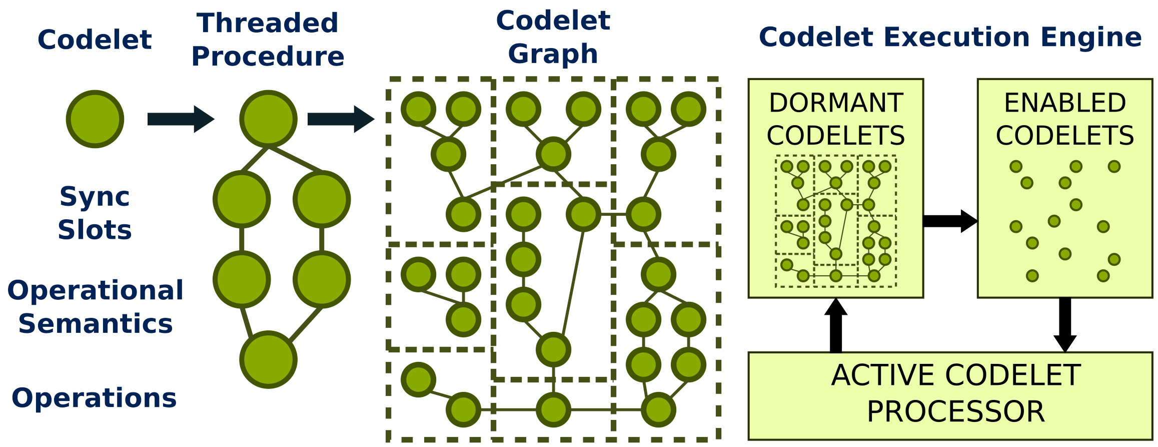

Figure 1: Codelet Model

can borrow and be interchanged with other dialects. For example, if an IR holds useful, source-level information about its matrix operations, a developer has the opportunity to create their own linear algebra dialect or even borrow from an already existing IR like the 'linalg' dialect integrated into MLIR. Consequentially, it is easy for developers to create their own custom domain-specific language or even an extension to an already existing language.

2.2 DEMAC

The Codelet Model for computation [11] [12] [13] is a hybrid Von Neumann / Dataflow Program Execution Model (PXM) that leverages traditional sequential architecture features while borrowing ideas from dataflow models of computation. We leverage the argument fetching dataflow to allow exploiting side-effect free parallelism at two different levels: Codelets and Threaded Procedures. As is the case in dataflow models, there is no strict order of computation of Codelets, instead they are executed as their structural or data dependencies are satisfied. While at the Codelet level it is possible to take advantages of features of sequential computation such as instruction level parallelism and out of order execution.

These features provide the operational semantics, and the expected behavior of the machine. The program is defined as a Dataflow graph, which is a set of nodes called Codelets interconnected by edges that represent data dependencies. Tokens are used to represent data that flows from a node that produces data to a node that consumes it in order to carry out operations, and they are stored in the Threeaded Procedure.

Codelets are the unit of Computation of the Codelet Model. They are event and data driven, which means their execution depends upon availability of the resources and data needed to start the computation.

The Codelet Model maintains the ordering constraints among instructions within the Codelet, while at the same time loosening the constraints between different Codelets. A Codelet is a sequential, non-preemptive, atomically-scheduled set of instructions. (1) Sequentially Executed:

Modern processors perform sequential execution very efficiently, even when there are many dependencies among the instructions. Features and control mechanisms present in this kind of processor provides desirable levels of performance without the need for additional, specialized hardware. (2) Non-Preemptive: Once a Codelet begins execution, it remains active in the CPU until it is finished executing. (3) Atomic Scheduling: A Codelet cannot be interrupted, so resources should be allocated exclusively for the execution of it.

Threaded Procedure (TP) consists of one or more Codelets that share local variables and input parameters of that TP. A Codelet is always part of an enclosing TP, much like contexts in modern programming languages. The tight coupling between Codelets within these categories requires a fine-grain level of synchronization and data sharing mechanisms.

Sync-Slots (SS) are used to make sure that all dependencies have been satisfied before a Codelet is enabled, a counter is used to keep track of tokens received. In a TP, local variables are used to transfer data from one Codelet to another. Sync-Signals are used as tokens. A Sync-Slot contains: A **Sync-Pointer (SP):** Binds the SS to one of the Codelets in the TP. A **Sync-Count (SC):** Indicates the number of dependencies to be received by the SS before the specified Codelet is enabled. A **Reset-Count (RC):** If SC reaches 0 the Codelet specified by the SP is enabled. After the Codelet execution has started, SC is set back to the RC. The use of a RC allows Codelets to be enabled multiple times.

Codelet Abstract Machine (CAM): The program is divided into small pieces (Codelets) so the machine can divide the workload among multiple cores. CAM's hardware is composed of a cluster of nodes that are interconnected by a network. Each node has multiple cores that can feature a heterogeneous architecture with specialized hardware to manage communication and synchronization among nodes/cores. The two-layer hierarchy of Codelets and TPs allows computation to be segmented into two levels of granularity. While TPs are assigned to a node, Codelets are assigned to cores in that node. At least two specialized processing units must exist: Scheduling Unit (SE) and Computational Unit (CU). SEs manage synchronization and tasking, CUs execute Codelets that are assigned and fired by a SE.

The behaviour of the machine is defined based on the different states of a Codelet. First, there is a pool of "Dormant" Codelets that are not ready to begin execution. When all dependencies have been fulfilled the Codelet changes to "Enabled" state, meaning it is ready to execute. Since CUs may be busy at that time, there may be a delay between the time a Codelet is "Enabled" and the time it fires and starts running (which occurs when it is assigned to a CU it becomes "Active"). Once the Codelet has been executed it becomes "Dormant" again, allowing it to be executed again. This mechanism is called the "Codelet Execution Engine", as represented in Figure 1.

When a procedure is invoked the system allocates and initialize the frame, enabling the initial Codelet that must wait for the machine to have sufficient resources available for execution. All other Codelets in the procedure rely on the machine to dynamically verify dependencies. SS are updated when a Sync-Signal is activated to alert the recipient that a specific control or data dependency has been satisfied.

Memory Model defined to support the Codelet PXM, must have the following properties: (1) All objects in the system which may be accessible by more than one TP must have a globally unique identifier. This includes: Frame identifier (FID), Codelet identifier (CID) and SS. Each instance of a codelet can be uniquely identified by a pair (FID, CID). (2) 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 a given stack pointer value has the same meaning on all such elements. (3) An active Codelet must have direct, low-latency access (through load and store operations) to both its private context and the frame it shares with other Codelets in the same procedural instance. (4) 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.

Each processor must be able to determine the exact location of any given memory reference. 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.This also allows the use of recursive calls. Because procedures are explicitly terminated, no garbage collection of frames is needed.

Operations provided as a set of primitives that may have a hardware or software implementation to support and manage interaction among processing units or between processing units and memory. The execution of a Codelet-based program relies on operations performing the following functions: (1) Invocation and termination of TPs and Codelets, (2) Creation and manipulation of SS, (3) Sending Sync-Signal to SS, either alone or atomically bound with data (can be initiated by the consumer or the producer)

3 Motivation & Problem Formulation

3.1 CODIR

In order to improve performance on Neural Networks, there has been an introduction of acceleration hardware. Commercially available General-Purpose GPU (GPGPU) allow the use of powerful multi-node architectures in desktop computers and clusters. Additionally, multiple neuromorphic chips inspired by dataflow models of computation have been proposed. Both approaches have proven to yield to considerable improvements. Although, when it comes to the integration of heterogeneous systems in general purpose computation, the coordination between different types of computational units usually involves additional effort for programmers to define the mapping of computation and hardware resources. Through the use of the Codelet Model, we expect to lift this burden from software-implemented runtimes and users.

The Codelet Model is an example of a program execution model and therefore requires hardware and software to agree upon a single structure and operation of parallel systems with intentions of achieving increases in Performance, Portability, and Productivity. It is possible to observe the utility of such an abstraction through viewing Instruction Set Architectures as a contract between software developers and hardware architects for sequential computing. By fixing the ISA, a disjoint evolution of hardware and software was allowed resulting in considerable progress in the two worlds. However, for parallel systems no such standards exist for a proper Program Execution Model. The use of the Codelet Model could provide potential benefits for achieving parallelism of data, pipelining of instructions, and also allowing for improved utilization of heterogeneous systems.

There has been substantial work on proving the Codelet Model as a feasible program execution model [1,10,17], many of which use DARTS. However, there are several limitations from a pure software implementation using C++. First, it is difficult to provide valuable arguments in favor of a hardware implementation. Second, it provides few opportunities to exploit compiler technology. While the former is tackled in the paper "DEMAC, A Modular Platform for HW-SW Co-Design" [18] co-published with this work, we aim to tackle the second problem by creating a software infrastructure of the Codelet Model API.

By utilizing MLIR, we propose a dialect that allows for the representation of programs in the form of Threaded Procedures and Codelets as explained in Section 2. This infrastructure will foster an environment suited for Codelet Model optimizations much like LLVM does with LLVM IR. For example, LLVM is able to apply the axioms that govern the fused-multiply-add operation to perform optimizations which may require switching the order of operands. Without the context of these axioms (commutativity, associativity, transitivity, etc.) the compiler would be constrained to treating this operation as a generic operation. If this was the case the compiler would not be able to perform the operand switching necessary for the given optimization. The MLIR compiler infrastructure provides ample opportunity for developers to include this underlying understanding of the domain one builds a compiler for. In creating our own CODIR Dialect, we will be able to integrate the concepts of the PXM into a compiler able to optimize Codelets while simultaneously decoupling the runtime to explore different implementations through code generation to either software or hardware targets.

To demonstrate the efficiency of this approach we target ML applications with a DSL that acts as a user of the PXM. The similarities shared between the Codelet Model and the human brain make it a suitable program execution model for AI applications. We intend to exploit the similarities between Neurons of a Neural Network and Dataflow based models of computation, similar to the work conducted by [1].

It becomes imperative to decouple the programming interface from the executing hardware in order to achieve this. Not only that would allow us to progress in hardware development without requiring a re-structuring of our programs, but it would allow us to explore compiler technology for the optimization of Codelet-based programs.

To explore hardware implementations of the Codelet Model we also propose The Delaware Modular Assembly Cluster (DEMAC), designed to integrate an array of embedded systems with a set of 3D-printed frames that provide support for the boards, the cooling units, and power units. Each board of the DEMAC cluster combines the resources of a dual-core processor, a 16-core coprocessor, and an embedded FPGA with the flexibility of a complete open-source stack. The mount is a low-cost implementation with a scalable, open-source structure designed to fit 4 units of a standard sized rack. Having multiple nodes allows us to explore distributed versions of the Codelet Model where there is no notion of shared-memory.

This work aims to solve two address two important questions in the research of the Codelet Model:

- What is the appropriate software infrastructure and abstraction to allow for Codelet programs to be defined, optimized and lowered to the hardware API?
- What is the appropriate methodology to design and implement such software infrastructure?

4 Contributions

4.1 CODIR

The main contributions of the paper are as follows: (1) Decouple program structure from the current API implementation of the Codelet Model (DARTS) to allow for an environment suitable for structural reorganization of programs, perform optimizations, and progressively lower programs to machine code (2) Describe some of the fundamental syntax and semantics necessary for an intermediate representation to unify high level languages representation of the Codelet PXM (3) Define a basic language with operations, types, and attributes required for representing a Dataflow program and the translation of high level languages to machine code or runtime system API.

4.2 DEMAC

The major contributions of this project are: (1) Designing and implementing the cluster as a platform to develop and test hardware-software co-design features for the Codelet PXM. (2) Developing the extension of the PXM for a distributed runtime, including the description of mechanisms and software implementations. (3) Developing hardware features using the FPGA to implement specialized modules that support the execution of programs defined with the PXM API. These contributions are milestones towards the main objective of research carried out in CAPSL group and are used by other members in different projects.

5 Solution Methodology

5.1 CODIR

The current implementation of the Codelet Model in the DARTS C++ library features a combination between runtime and Codelet Model API. We seek to decouple these two elements

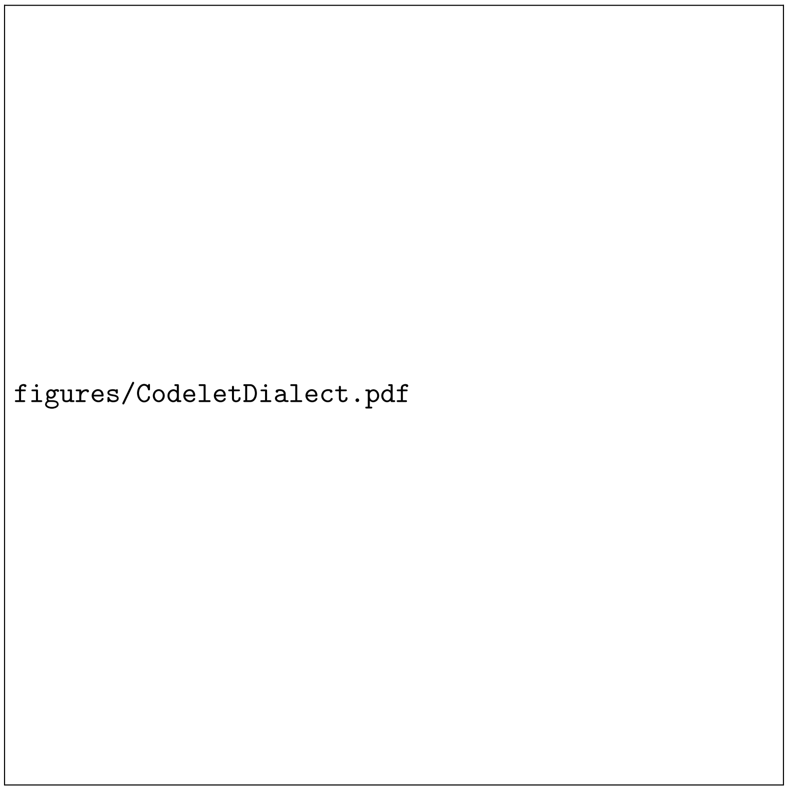

Figure 2: Codelet Dialect Software Infrastructure

while allowing reusability of the API into other implementations of the runtime model using the MLIR compiler infrastructure. This infrastructure is presented in Figure 2. There are three major parts in the infrastructure. The Hardware, the Runtime Systems and the MLIR front end. At the hardware level, we map the Codelet Abstract Machine parts. The different Colors of the Computational Units represent the potential for heterogeneity. There exists two possible paths for a runtime implementation: An LLVM based implementation that connects to a stripped down version of the DARTS runtime system as a linked library during the computation as well as a hardware software co-design implementation (e.g. the DEMAC cluster). Finally, the MLIR shows the definition of a Codelet Dialect that allows for codelet optimizations and code generation. It also demonstrates the connection to other dialects such as the Tensorflow Dialect, for progressing lowering of higher level domain specific languages.

5.1.1 CODIR: MLIR Dialect

The Codelet Model Intermediate Representation (CODIR) is our proposed dialect to create a compiler and domain-specific language for the Codelet Model. We believe MLIR is the compiler infrastructure to build this upon due to its adoption across the industry and its design rationale matching our goals exactly. These types will have to emulate the same properties, respectively outlined in the Codelet Program Execution Model. Codelets, for example, will have to be non-preemptive, atomically scheduled, and be side-effect free. Additionally there should be some basic structures and functions for the Codelet type. First, a Codelet should have a state which can change between ready, enabled, dormant, and firing and should be visible to the enclosing TP. Based on this state, the Codelet could execute the fire function which is abstracted from the user. This function should consist of three parts: (1) Consume input tokens and read from memory. (2) Execute the instructions contained in the Codelet. (3) Write results to memory and place tokens on the corresponding output arcs. The user should only have to manage defining the inputs and outputs along with designing the code to be executed when the Codelet fires. The Codelets should have a respective ID such that it can be easily referenced and properly debugged. MLIR aids in the debugging process by requiring location information throughout the definition of a Codelet. To this end, it will be trivial to search for a point of error during compilation so long as an instance of a type has a unique ID. The Codelet type should also have a synchronization slot (sync slot) structure to store information about its dependencies including the number of tokens currently present in its input arc(s) and adjust its state accordingly. Beyond just Codelets, there must exist a Threaded Procedure type as well. This type should function similarly to that of a task in OpenMP in that it should be declared with an input list and output list aimed at addressing problems involving data races. The analysis and prevention of data races at the compiler level can be incredibly difficult and not always necessary so we leave their management up to the user. The example below illustrates a potential implementation of our language using a directive-style extension to C-family languages much like the OpenMP Platform.

```
1  /* Fibonacci Example */
2  #pragma codir tp id{"fib"} in(int n) out(result) {
3     fib(int n, int &result, SYNC done){
4        /* TP frame attributes */
5        int x, y;
6        /* Create Codelets */
7        #pragma codir cod id("check") dep(0,0) {
8           if (n < 2) {
9              SYNC(n, result, done);
10          } else
11              /*Asynchronous Calls*/
12              x = fib(n-1, &x, adder);
13              y = fib(n-2, &y, adder);
14          }
15       }
16       #pragma codir cod id("adder") dep(2,2) {
17          /* result is the token
18             and done is the sync slot */
19          SYNC(x+y, result, done);
20       }
21    }
22 }
```

- **Line 1**: The `tp()` directive denotes the enclosed code as a threaded procedure declaration which can be referenced through the `id("fib")`. The `fib` TP takes in an integer `n` as input, outputs to a reference to an integer result in another TP, and upon completion of the `tp`, signals the `done` Codelet through its respective **sync slot**

- **Line 3** Declares two variables, `x` and]`y`, local to the TP frame and available to the inner CDG.
- **Line 6** Instantiates a `check` Codelet. This Codelet uses the argument `n` of the TP and either spawns two more `fib` TPs or signals back the caller for continuation.
- **Line 9** If `n` is less than 2, the recursion ends and the value is sent to the caller TP. the `SYNC` operation assigns the value of `n` to the `result` location in the caller TP, and signals the codelet referenced by `done`, which is also located in the caller TP.
- **Lines 12 & 13** If `n` is greater than two, recursively spawn two `fib` TPs with $n-1$ and $n-2$ as the input number respectively. The new TPs are asynchronously executed. When their execution finishes, the results are sent to `x` and `y` respectively. These two variables represent the data of the token that is needed by the `adder` Codelet. The `adder` Codelet is sent in order to allow the callee to signal back when the data is available. In reality the signaling mechanism is done through the sync slot.
- **Line 16** Instantiate an additional Codelet `adder` which takes in the values written to memory locations `x` and `y` available in the frame TP, and writes their summation to the `result` reference of the caller TP. The `SYNC` function in Line 19 also triggers the continuation codelet of the caller TP.

There are many more aspects we still must consider at length to ensure the most intuitive, efficient, and scalable Codelet Model domain-specific language. After writing in our DSL, the users code will go through several stages of progressive lowering to intermediate representation, with the most relevant one being the CODIR Dialect. Because of the nature of MLIR, we will be able to provide the compiler with source level information about the Codelet Graph, thus leading to Codelet Model specific optimizations with context similar to LLVM's understanding of the add operation previously mentioned. Following CODIR, we will have the options to lower to LLVM IR and link to a runtime environment built into LLVM or lower to DEMAC IR such that a hardware-software codesign runtime environment can be explored.

5.1.2 Neural Networks to CODIR

Given the data-driven and event-driven nature of data-flow inspired execution models, it is possible to observe similarities between Codelets and Neural Networks. Three characteristics that are directly comparable are as follows. (1) The execution of both Codelets and Neurons only depends on their inputs. There are no side effects on the execution of a Codelet other than its outputs. (2) The activation function of a neuron is comparable with the operation that is described inside a Codelet. (3) The direct acyclic graph formed by a Codelet graph is flexible enough to be able to represent a neural network without requiring any modification.

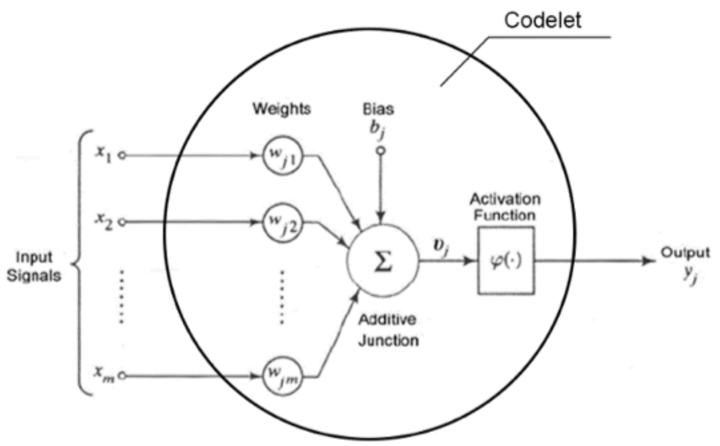

Figure 3: Mapping Neurons to Codelet [1]

However, there are other aspects that need extra consideration. For example, information in a neural network is stored in a distributed way. In the Codelet Model. In order to achieve this, the distribution of weights and data of the NN can be mapped to the concept of *Threaded Procedures*, as it also stores the context for the execution of *Codelets*.

5.1.3 Protocol Buffers to DARTS

Previous work in [1] showed the benefits of mapping Neural Networks to the Codelet Model, however the LeNet implementation of this paper was written manually. We are working on a prototype transpiler that, based on Neural Networks described through protobuffer, it generates an implementation in DARTS C++ code interface [19]. Preliminary results for this framework will be included with the final version of this manuscript. The approach is to implement the operands in the NN in DARTS (e.g. Conv2D and MaxPool) as well as the concept of Tensors. Then, we generate the mapping of the NN connections and context as threaded procedures and signals between Codelets, resulting in an implementation of the NN in the Codelet Program Execution Model.

The transpiler is implemented through Python's Tensorflow library, which 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. The programmer has all of the necessary information for generating Codelets. The lack of ability to spawn special 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 [20], 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.

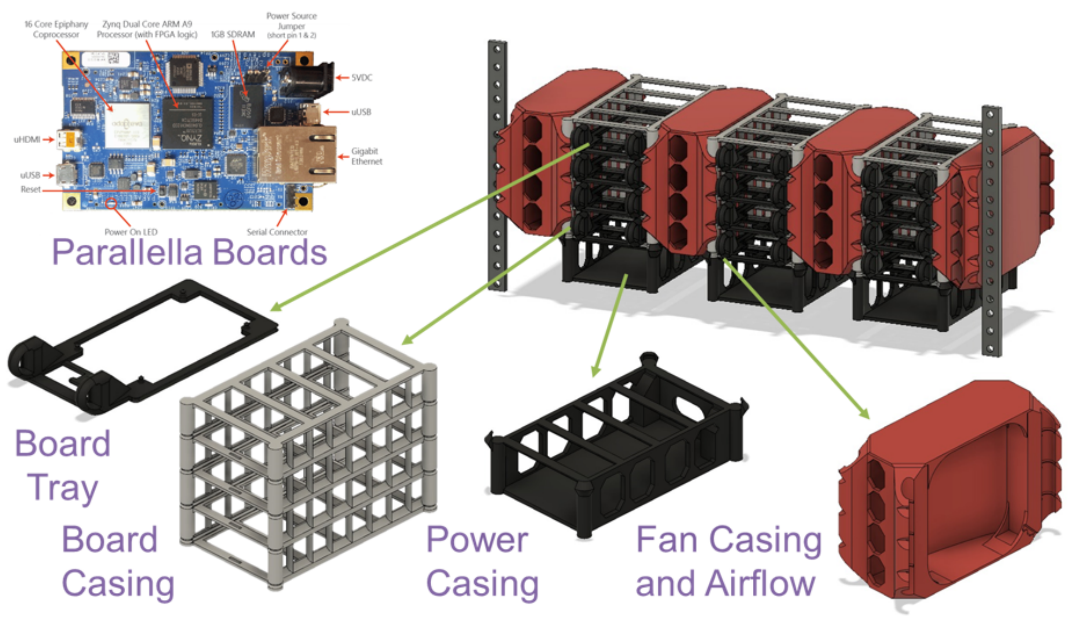

Figure 4: DEMAC: Delaware Modular Assembly Cluster

5.2 DEMAC

5.2.1 Hardware

To support developing and testing hardware and software features based on the Codelet PXM, we propose a platform composed of multiple nodes interconnected by a network. Each node is a multi-core off-the-shelf embedded system that provides a Linux-based environment. Additional requirements are for the system to include an FPGA and an open-source stack to support further development.

DEMAC: The Delaware Modular Assembly Cluster is designed to integrate an array of embedded systems with a set of 3D printed frames that provide support for the boards, the cooling units, and the power units. Each board [21] combines the resources of a dual-core processor, a 16-core coprocessor, and an embedded FPGA with the flexibility of a complete open-source stack. The mount is a low-cost implementation with a scalable structure designed to fit 4 units of a standard size rack. Files for the frames are open source. Having multiple nodes allows us to explore distributed versions of the Codelet Model where there is no notion of shared-memory.

DEMAC has 5 different frames:

- **Board Tray:** Holds a board and a label with the name of the node (NOPA##)
- **Board Casing:** Provides housing for 4 Board Trays
- **Power Casing:** Holds usb hub to power 4 boards

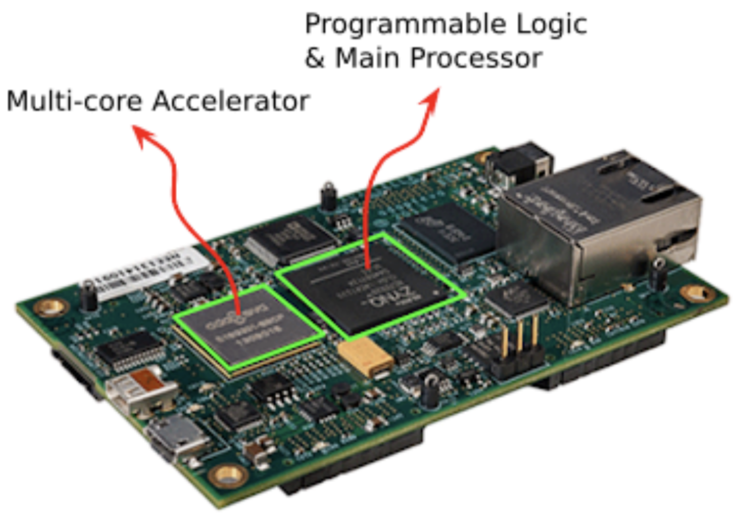

Figure 5: Embedded system used in DEMAC

- **Fan Casing:** Air input and output are designed to allow a fan to independently cool down a whole Board Casing

- **Cooling Auxiliary Casing:** Necessary to complete the structure to be mounted on a rack

The cluster is mounted on top of a switch that provides an ethernet connection between the nodes and the HeadNode. Each board is connected to the switch using an RJ-45 cable. The HeadNode allows users to interface with the nodes. 12 Board Trays, installed in 3 Board Casings with their respective power supplies and interconnected by 3 Fan Casings and a Cooling Auxiliary Casing are used to build a structure that fits a standard rack. Power for the boards is independent from the auxiliary systems (Cooling and Network).

Developing hardware-level features for the Codelet Model is possible by using the on-chip FPGA. The many RISC Cores of the coprocessor can be used as SUs or CUs, and the lack of cache coherency allows us to explore more flexible memory models as well as different mapping mechanisms for the Codelet Abstract Machine. The low cost of the system and its open-source nature promotes interdisciplinary collaborations and expansion of parallel computing to other fields (e.g. Artificial intelligence, molecular dynamics, robotics).

Embedded System board used is affordable, highly parallel, and open source. It features a programmable logic/main processor combined chip and a multi-core accelerator. The accelerator operates on a RISC architecture and is used to offload computation for acceleration. The accelerator, together with its low cost and the openness of its software stack, is what 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 programmable logic implemented on the FPGA.

Each core 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 instructions that allow atomic access. The user is responsible for managing this memory and using it for code and data. Programming the system is possible using C, assembly language, and some limited functionalities of C++. However, all of the software stack (i.e. operating system, drivers, libraries, and FPGA code) is completely open source.

Thanks to these properties, the system is highly flexible and customizable. Its barebones 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 execution of this project more challenging.

5.2.2 Software

Each board runs a customized linux distribution, allowing us to use this embedded system as a computer connected to a network. The current implementation includes OpenMP and MPI on the dual-core processor of each node. While OpenMP is used to communicate and synchronize the cores on the main processor in a single node, MPI provides the mechanisms to control and share data among the different nodes through the network. Their main advantages are portability, ease-of-use and a clearly defined set of routines. 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.

CODIR: Codelet Dialect for IR. In order to decouple the API of the Codelet Model from the runtime, and to allow the use of compiler technology at the Codelet level we have defined an MLIR dialect that represents operations and computational elements of the Codelet Model. Such approach would also allow for mapping to different hardware and software based implementations of the runtime, providing re-usability of infrastructure and optimizations as well as progressive lowering of Codelet Code into runtime executable instructions. The details of this work are presented in the paper ”CODIR: Towards an MLIR Codelet Model Dialect” [**?**] to be submitted in conjuntion with this work.

eSDK is a set of software tools [**?**] provided by board developers to configure and interface with its hardware features. This includes compiler, debugger, assembler, libraries linker and loader to interact with the coprocessor. The first four are used to write code to execute in the accelerator, the assembler is used to define low level functions. The linker and loader describes the mechanisms that deal with memory management, this can be modified to define allocation policies. Functions defined in the libraries are used to manage memory allocation and computation execution of cores in the accelerator.

5.2.3 Runtime

Dataflow runtimes carry out computation of programs represented as a direct acyclic graph where nodes are sets of instructions and edges define data dependencies, rules define that nodes can begin execution only if all data dependencies have been met. There is no specific order in the execution of Codelets in a program, only that given by the data as it becomes available.

Legacy Runtimes are previous implementations that provide a base-ground to define the mechanisms required to support a dataflow runtime. EARTH model [22, 23] describes the functionality of the elements that are essential in this kind of architecture to schedule and execute Dataflow programs. In Addition to describing the architecture and operations, it provides software tools as a C-library and a hardware implementation: The EARTH-MANNA Machine [24].

Another remarkable project that includes hardware implementation is Cyclops-64 [25], Implemented as an accelerator with a Dataflow-based runtime, computation is offloaded from a cluster with a conventional architecture into a computational array of nodes with multiple cores. Sections of programs can be executed by a myriad cores, ruled by dataflow semantics in a specialized system that reduces the overhead caused by conventional OS to carry out computation.

The first one describes an implementation of a Dataflow machine that includes hardware features to manage communication and synchronization among nodes, while the second one is more similar to an accelerator, where specialized hardware is attached to a host system that provides an interface to offload computation.

DARTS: Delaware Adaptive Runtime System [15, 17], is a software implementation for a multi-core, single-node system. It is written in `C++` and distributed as free and open-source software. Threaded Procedures (TP) and Codelets are defined as class objects at compile time and are instantiated at runtime. A TP is instantiated when a Codelet performs a TP invocation. This operation creates a TP handle and sends a request to the runtime to allocate memory for the TP instance and the instances of the Codelets it contains. Once this is done, Codelet instances are executed upon satisfaction of their dependencies.

A two-level scheduling mechanism, comprising TP schedulers and micro-schedulers, is used for the scheduling and execution of Codelets. A TP scheduler is a core which has been assigned the role of SU and has a *TP queue* (TPQ) with TP instantiation requests and a *ready-Codelets queue* (RCQ). A *microscheduler* is a core acting as a CU with a Local Codelet queue (LCQ), which is fed by its local TP scheduler with Codelets from its RCQ.

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 [26], 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

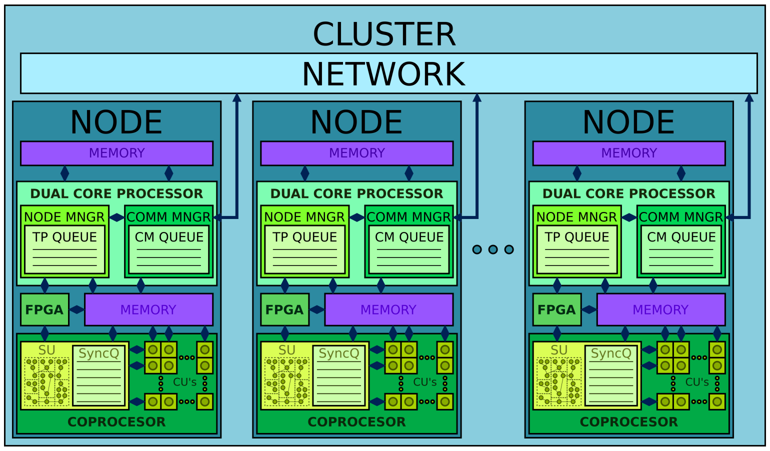

Figure 6: DECARD Abstract Machine Model

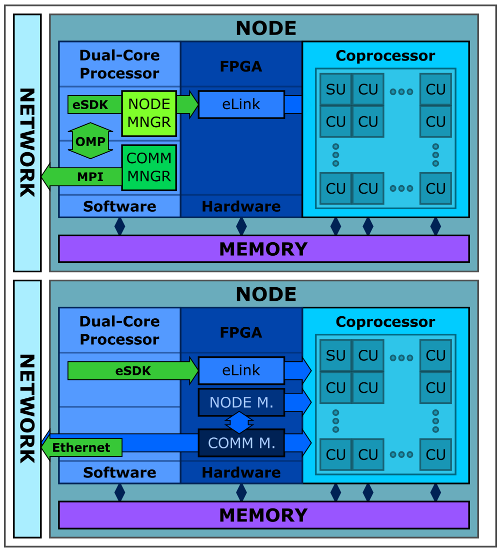

Figure 7: HW-Software Co-design

- Synchronization mechanisms (Mutex, Barrier)
- Memory allocation and management: 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 (CU): Used for designating a given core to be a core for carrying out computation.
- Scheduling Unit (SU): 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

DECARD: Distributed Execution Codelet Accelerated Runtime for Dataflow, allows task scheduling and communication of a multi-node system. Providing a distributed-global shared-address space requires mechanisms that allow processing elements to access memory from different nodes. To implement these, DEMAC features two elements with the following functions:

- Node Manager: 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 across TPs and Data) and allocating them in local memory.

The upper part of Figure 7 represents a software-based implementation, while the lower part represents a hardware implementation of modules that carry out these functions. In the first one, the Node Manager uses the libraries provided with the eSDK to schedule and manage tasks in the accelerator and relies on OpenMP to share information with the Node Communicator. The Node Communicator uses MPI functions to send and receive information to/from other nodes in the system.

On the hardware version, these two modules are implemented on the FPGA. The communication manager interfaces directly with the network through some of the chip's I/O pins, allowing having multiple ports to create a more interconnected network structure in some sort of 3-dimensional array. The node manager administrate TP Queues and hardware resources to execute computation and interfaces with the communication manager to send/receive information to/from other nodes.

6 Related Work

6.1 Codelet Model LeNet5

The Codelet Model implementation of the LeNet NN by Zeng et. al. [1] shows hopeful results for this PXM. When analyzing a parallel model, superior computational efficiency and scalability are top priorities for success. Efficiency referring to the ability to achieve high utilization of the available hardware resources, and scalability referring to the ability to maintain performance as the number of available computing units increases. The authors used LeNet as a benchmark ML application for comparing the efficiency and scalability of the Codelet Model when compared to current popular frameworks and execution models such as `OpenMP`. LeNet5 is considered a classic NN used for digit recognition. Two versions of the network created: A pure DARTS implementation and a TensorFlow-embedded DARTS design. Running the network using the DARTS implementation shows promising speedup over OpenMP as the number of threads was increased. When comparing TensorFlow to a TF-DARTS model it too showed improved performance for higher core counts.

6.2 MLIR Dialects

MLIR's dialect construct has been integrated into several successful projects already. One example is the Intermediate Representation Execution Environment (IREE) [27]. IREE functions as an end-to-end compiler for lowering intensive ML applications down to a unified representation for real-time inference against hardware accelerators. The goal of the project is to encode both the scheduling logic used for dependency communications and the execution logic into hardware and API-specific binaries. IREE is compilation-based and therefore does not map ops to their kernel representations making it There does not exist an IREE runtime environment as it is compilation-based and does not map ops to their respective kernel implementations. Consequentially, there does not exist a runtime environment for IREE. The dialect relies on Tensors as their main datatype which differs from the approach we aim to take; having Codelets and Threaded Procedures being the main types. Additionally, we plan on creating a full runtime environment for the Codelet model, further differentiating our visions.

Intel has also developed their own dialect, PlaidML [28], with the intention of making deep learning applications executable across any hardware. They leverage MLIR to make the integration of new software and hardware into their compiler stack as well as taking advantage of the easily implemented optimizations available. In addition, they have created a custom C++/Python embedded domain-specific language known as EDSL to promote ease of use for developers wanting to use PlaidML. The emphasis of this project is placed on extensibility across all different types of hardware including desktop/laptop computers, embedded systems, mobile phones, and server technology. Our project also aims to be portable across a range of hardware and accessible to developers though while additionally providing numerous benefits over many traditional program execution models. We intend to unify the PXM of the system as a whole, and then use ML as a target application that demonstrate the need for such abstraction.

7 Preliminary 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: (1) 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. (2) 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. (3) 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.

8 Future Work

8.1 CODIR

Future work for the CAPSL group will consist of the development of the CODIR Dialect along with a runtime environment to be created within LLVM. In accomplishing this we will have created an alternative to other popular frameworks including TensorFlow, PyTorch, mxNET, and many more. Having this as a resource would provide a direct opportunity to compare the viability of the Codelet Model over other program execution models. An additional benefit is providing accessible programmability of the Codelet Model.

8.2 DEMAC

This paper defines the necessary mapping between the Codelet Program Execution Model of computation, and the DEMAC infrastructure. However, as expected from system design, there is still a lot of details to narrow down. Current implementation of DARTS lacks the needed mechanisms for distributed computation. We have started to define the mechanisms through MPI to achieve this. eDARTS implementation is also on the way and there is still 20% left to be done. Perhaps the most challenging part of it is the FPGA implementation, which requires us to modify the current implementation. Such implementation is openly available, but the lack of good documentation has made the process difficult. Currently we have achieved generating the original bitstream from scratch, and we have started including new designs together with the already existing one. There is also a need to do the necessary modifications in order to create the communication between the Linux kernel and the modified FPGA version.

9 Conclusion and Future Results

9.1 CODIR

We conclude that creating the CODIR dialect using the existing MLIR infrastructure to create a domain-specific language along with a compiler for the Codelet Model would grant users access to the countless benefits associated with the underlying execution model. The model's efficacy has been demonstrated already through the research conducted by many different groups for a variety of projects. For this reason, we are confident progressing the accessibility of the model will provide promising future results when comparing to other frameworks and program execution models. The dialect would act as a direct software interface with the model through the domain specific language. Additionally, the progressive lowering promoted by the MLIR infrastructure would permit a custom compiler to apply knowledge specifically pertaining to the axioms governing Codelets. Finally, then lowering to a plethora of hetergeneous hardware targets with intentions of applying our future work to the DEMAC cluster [18] ultimately providing an opportunity to lower code to a hardware-software codesign implementation of the

Codelet Model.

The gap between software frameworks and machine code generation will only continue to grow if a standard program execution model governing the contract between a program's description and how it will be executed on the target hardware continues to not exist. We believe the Codelet Model is this model and through continued development of our proposed Software Infrastructure we are confident the viability of the model will not go unnoticed.

9.2 DEMAC

We can conclude that having this kind of platform to support HW-SW co-design is fundamental to implement and evaluate features that can increase the synergy of the Codelet PXM with a physical machine. The flexibility and versatility of this system allows us to better understand current computational capabilities and limitations regarding traditional programming models. This project provides an environment where we can define computation in a broad way and extend the abilities of these kind of systems.

Acknowledgment

We would like to thank the anonymous reviewers of the paper. We gratefully acknowledge the assistance provided by the Argonne Computational Scientists. This work is partially supported by the National Science Foundation, under award SHF-1763654. Additionally we would like to commend the dedication and unwavering efforts put forth by Dr. Guang R. Gao.

References

[1] S. Zeng, J. M. Monsalve Diaz, and S. Raskar, "Toward a high-performance emulation platformfor brain-inspired intelligent systemsexploring dataflow-based execution model and beyond," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 628–633, 2019.

[2] E. A. Lee, "The problem with threads," *Computer*, vol. 39, p. 33–42, May 2006.

[3] "Cerebras Wafer Scale Engine: An Introduction." `https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-An-Introduction.pdf`.

[4] L. Deng, G. Wang, G. Li, S. Li, L. Liang, M. Zhu, Y. Wu, Z. Yang, Z. Zou, J. Pei, Z. Wu, X. Hu, Y. Ding, W. He, Y. Xie, and L. Shi, "Tianjic: A unified and scalable chip bridging spike-based and continuous neural computation," *IEEE Journal of Solid-State Circuits*, pp. 1–19, 2020.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[8] P. Qu, J. Yan, Y.-H. Zhang, and G. R. Gao, "Parallel turing machine, a proposal," *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 269–285, 2017.

[9] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," 2020.

[10] G. Gao, J. Suetterlein, and S. Zuckerman, "Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond." Technical Memo, April 2011.

[11] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a ?codelet? program execution model for exascale machines," in *In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT ?11*, p. 64?69, 2011.

[12] J. S. G. Gao and S. Zuckerman, "CAPSL Technical Memo 104: Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond," April 2011.

[13] G. R. G. K. L. Rishi Lee Khan, Daniel Orozco, "Codeletset representation, manipulation, and execution-methods, system and apparatus," May 2012. US Patent WO2012067688A1.

[14] J. B. Dennis and G. R. Gao, "An efficient pipelined dataflow processor architecture," in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, (Los Alamitos, CA, USA), pp. 368–373, IEEE Computer Society Press, 1988.

[15] S. Joshua, "Darts: A runtime based on the codelet execution model," Master's thesis, University of Delaware, Newark, DE, 2014.

[16] CAPSL, "Darts source code." `http://www.capsl.udel.edu/codelets/downloads/darts_20150318.tar.gz`.

[17] J. Suettlerlein, S. Zuckerman, and G. R. Gao, "An implementation of the codelet model," in *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, (Berlin, Heidelberg), pp. 633–644, Springer-Verlag, 2013.

[18] J. M. M. D. S. R. D. F. Diego A Roa Perdomo, Ryan Kabrick, "Demac, a modular platform for hw-sw co-design," Technical Memo 136, University of Delaware, April 2020.

[19] S. R. J. M. M. D. G. R. G. Diego Roa, Ryan Kabrick, "Brain-flow : A brain inspired data-flow implementation using demac," Technical Memo 134, University of Delaware, October 2019.

[20] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multitask cascaded convolutional networks," *IEEE Signal Processing Letters*, vol. 23, p. 1499–1503, Oct 2016.

[21] Parallella, "Parallella board." `https://www.parallella.org/`.

[22] H. Hum, "A design study of the EARTH multiprocessor," *in Proceedings of Parallel Architectures and Compilation Techniques*, 1995.

[23] H. Hum, X. Tang, Y. Zhu, G. Gao, X. Xue, H. Cai, and P. Ouellet, "Compiling C for the EARTH multithreaded architecture," in *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pp. 12–23, Oct 1996.

[24] H. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao, and L. Hendren, "A study of the EARTH-MANNA multithreaded system," *International Journal of Parallel Programming*, vol. 24, no. 4, pp. 319–348, 1996.

[25] Y. Niu, Z. Hu, K. Barner, and G. Gao, "Performance Modelling and Optimization of Memory Access on Cellular Computer Architecture Cyclops64," in *Network and Parallel Computing, IFIP InternationalConference, NPC 2005, Beijing, China, November 30 – December 3, 2005, Proceedings* (H. Jin, D. A. Reed, and W. Jiang, eds.), vol. 3779 of *Lecture Notes in Computer Science*, pp. 132–143, Springer, 2005.

[26] G. G. Jose Monsalve, "Parallella implementation of the codelet model." `https://www.capsl.udel.edu/codelets_parallella.shtml`, 2018.

[27] Google, "Iree: Intermediate representation execution environment." https://github.com/google/IREE, Apr 2020.

[28] Plaidml, "plaidml/plaidml." https://github.com/plaidml/plaidml, Apr 2020.