Published in final edited form as:
IEEE Micro. 2022 ; 42(5): 89–98. doi:10.1109/mm.2022.3189416.

Yin-Yang: Programming Abstractions for Cross-Domain Multi-Acceleration

Joon Kyung Kim[★], **Byung Hoon Ahn**[★], **Sean Kinzer**[★], **Soroush Ghodrati**[★], **Rohan Mahapatra**[★], **Brahmendra Yatham**[★], **Shu-Ting Wang**[★], **Dohee Kim**[†], **Parisa Sarikhani**[‡], **Babak Mahmoudi**[‡], **Divya Mahajan**[§], **Jongse Park**[†], **Hadi Esmaeilzadeh**[★]

[★] UC San Diego

[†] KAIST

[‡] Emory University

[§] Microsoft

Abstract

FPGA accelerators offer performance and efficiency gains by narrowing the scope of acceleration to one algorithmic domain. However, real-life applications are often not limited to a single domain, which naturally makes *Cross*-Domain *Multi*-Acceleration a crucial next step. The challenge is, existing FPGA accelerators are built upon their specific vertically-specialized stacks, which prevents utilizing multiple accelerators from different domains. To that end, we propose a pair of dual abstractions, called Yin-Yang, which work in tandem and enable programmers to develop cross-domain applications using multiple accelerators on a FPGA. The Yin abstraction enables cross-domain algorithmic specification, while the Yang abstraction captures the accelerator capabilities. We also develop a dataflow virtual machine, dubbed XLVM, that transparently maps domain functions (Yin) to best-fit accelerator capabilities (Yang). With six real-world cross-domain applications, our evaluations show that Yin-Yang unlocks 29.4× speedup, while the best single-domain acceleration achieves 12.0×.

Keywords

Compilers; Runtime Environments; Hardware/Software Interfaces; Heterogeneous (Hybrid) Systems; Reconfigurable Hardware

I. Introduction

FPGAs have emerged as a promising acceleration platform for diverse application domains both at the edge and on the cloud (Amazon F1 instances [1] and Microsoft SmartNICs [2]). Despite the benefits, the accelerators by definition limit the scope of acceleration to an algorithmic domain, while real-life applications [3–5] often extend beyond a single domain. It is evident that for such cross-domain applications, utilizing multiple accelerators–even on a single FPGA–from different domains can unlock new capabilities and offer higher performance and efficiency. However, each accelerator often comes with its own vertically-specialized domain-specific stack, as illustrated in Figure 1(a), which by design is difficult to conjugate with other stacks. Thus, there is a need for a *horizontal programming abstraction* that enables programmers to develop end-to-end applications without delving into the isolated accelerator stacks.

To that end, this paper sets out to devise such abstractions by building upon a collection of programmer-transparent layers. We first devise a pair of dual abstractions, called Yin-Yang, where 1) the Yin abstraction allows *domain experts* to concisely describe the capabilities of each domain, and 2) the Yang abstraction enables *hardware designers* to abstractly denote compute capabilities and data interfaces for their FPGA accelerators, henceforth referred to as *engines*. The Yin abstraction also offers a lightweight programming interface that allows *programmers* to aggregate Yin-defined cross-domain capabilities together as a single program, while preserving the domain boundaries. Then, to enable the two abstractions to work in tandem, we develop XLVM (Accelerator-Level Virtual Machine) and its execution workflow is delineated in Figure 1(b). XLVM is a dataflow virtual machine that builds and executes the program as a Queued-Fractalized Dataflow Graph (QF-DFG). In QF-DFG, like fractals, each node is another QF-DFG but at a progressively finer granularity, until a node is a primitive scalar operation. For given QF-DFG, the XLVM's Engine Selector chooses components of the application (i.e., nodes of QF-DFG) to appropriate engines, using the engine specifications from Yang abstraction. XLVM also comes with Engine Compiler that compiles the individual engines into runnable executables and links them as a unified execution flow by automatically converting dependencies (i.e., edges of QF-DFG) to inter-engine communication between FPGA accelerators.

We collect diverse real-world applications and offer it as an open-source benchmark suite for cross-domain multi-acceleration. These applications range from deep brain stimulation, geological exploration, film captioning, stock exchange, medical imaging, and surveillance. Each of these benchmarks comprises algorithms from more than one domain where each is accelerated across multiple domain-specific accelerators. Using this benchmark suite, we evaluate the proposed abstractions and its concrete system implementation. By enabling cross-domain multi-acceleration, our work improves speedup from 12.0× to 29.4× (i.e., 145% extra benefits on average) compared to an end-to-end execution with a single FPGA accelerator that offers the highest gain. These results suggest the effectiveness of the Yin-Yang abstractions and their associated system framework in enabling cross-domain multi-acceleration.

II. Yin Abstraction

A. Abstract Domain Description

The goal of Yin abstraction is to delineate the capabilities of each domain, without any accelerator or application specific constructs. For every domain, a unique abstract definition, called domain description, is provided independently by the domain experts to pre-define domain's common capabilities. As the objective is to allow multi-acceleration, a domain description consists of a set of capabilities, each of which is a potential agent for acceleration. However, note that the capabilities defined in domain descriptions are accelerator-agnostic and not linked to a specific accelerator. Yet, these capabilities can be mapped to various accelerators through our virtual machine XLVM (Section IV). To enable programmers to use the domain capabilities in their applications, we also provide a set of lightweight programming model (Section II-B). Using the programming interface, programmers can develop their applications by importing the domain descriptions and instantiating the domain capabilities, while still preserving the domain specificity, interface, and boundary of each instantiation.

Example domain description for digital signal processing.—Figure 2(a) illustrates an example domain description for the digital signal processing domain. The domain descriptions are composed of the following: domain name, a set of capabilities with input/output semantics, a default reference implementation, and a cost model. The domain name is specified using the keyword **domain** on line 1. On line 4–22, the domain capabilities and their input/output are specified. For instance, lines 16–22 define the convolution capability, a frequently used operation in digital signal processing and is often accelerated by DSP accelerators. The **capability** keyword denotes the computational capability supported in this domain, and is followed by a unique denotation. Each capability has required input (**input**) and output (**output**) specifications as the arguments to its definition. The input and output data type and dimensions form the interface to the computation capability. We also have the **state** keyword that semantically stores the state across multiple executions. State is necessary for domains that share a temporal component such as robotics, data analytics, and deep learning. The **param** keyword denotes data whose values remain constant across executions.

B. Component & Flow Programming Model

The domain descriptions define the capabilities of individual domains that constitute end-to-end applications, but there is a need for programming interface that enables programmers to use the capabilities for application development without concerning the low-level hardware details. Due to the modular nature of the Yin abstraction, the Component and Flow programming model (CNF) is built upon lightweight annotations to create the linkage between domain description capabilities and end-to-end application kernels. Components and Flows in CNF represent the computation and dataflow in between, respectively. In particular, **Component** is a language construct that is explicitly used within the code, whereas the Flow is implicitly present in between.

Deep brain stimulation.—To demonstrate the use of CNF programming model, we take a cross-domain application, *deep brain stimulation* [5], as an example. This application takes and processes the electrophysiological response of the brain (DSP) to measure the biomarkers (Analytics), and generates a set of optical stimulations (Control) for memory enhancement in rats. In this setup, the electrophysiological activity of the brain is collected in real-time, and passed through Fast Fourier Transform (FFT) and a set of Band Pass Filters (BPF) of distinctive frequency bands (i.e., delta (0.5–4 Hz), theta (4–8 Hz), alpha (8–12 Hz), beta (12–30 Hz), and gamma (30–100 Hz)). Next, the pipeline uses Logistic Regression (LR) to decode and classify these brain waves to be used as biomarkers. Based on the classification output, a Model Predictive Control (MPC) process configures the synthesized brain waves (i.e., amplitude, frequency, and duration). Offloading the major compute-heavy algorithms to the corresponding accelerators–FFT to DeCO [6], logistic regression to Tabla [7], and control optimization to RoboX [8]–will provide runtime performance improvements[1].

Example CNF code for deep brain stimulation.—Figure 2(b) illustrates a CNF implementation of deep brain stimulation. First, lines 1–3 **import** domain descriptions into the application and bring the available pre-defined capabilities. Then, CNF enables programmers to express 1) the component boundaries, 2) its interfaces, and 3) the hierarchical structure. On line 5, the components are defined using the keyword **Component** along with its inputs (e.g. wave) and outputs (e.g. stimuli), which is followed by the code for its computation on lines 7–19. CNF also allows the programmer to express arbitrary levels of component hierarchy, where Components may be defined inside a Component. Once a component has been defined, the programmer can instantiate it as many times as needed to express the algorithm.

III. Yang Abstraction

A. Abstract Engine Specification

To manage various accelerators and to allow flexible additions of newly developed accelerators, we devise the Yang abstraction, as the counterpart to the Yin abstraction. The Yang abstraction offers a means for the accelerator developers to abstractly describe the accelerator specifications. In this paper, an *engine* denotes an abstract compute platform, which exclusively supports a single domain and is able to serve a subset of the capabilities defined in the corresponding domain description. Thus, Yang abstraction allows the engine developers to specify the provided capabilities and communication interfaces of an engine as a structured specification, called engine specification.

Example engine specification describing an digital signal processing accelerator.—Figure 2(c) illustrates an example engine specification of DeCO [6] using our engine specification language. To provide a flexible abstraction that can be used by a variety of engines, but also to ensure multi-acceleration, an engine specification needs to express 1) its own capabilities, and 2) the interface it exposes to connect with different engines. Line 3 shows that the engine name is specified using the keyword **engine** and domain (digital_signal_processing). On Line 4–7, how the engine communicates its input and output with the outside world is specified using the keyword **interface**. The engine specification provides pre-defined interfaces such as FIFO, SRAM, or BRAM, etc. Also, its capabilities (i.e., fft and band_pass_filter) and their semantics for input, output, weight, and configuration memory are specified in lines 9–19.

B. Hints for Engine Selection

Our Yang abstraction also offers two keywords, **fusion** and **cost**, to allow the engine developers to provide engine-specific information, which can be used as *hints* for the

[1]The original work [8] proposed RoboX as an ASIC but it is straightforward to develop the architecture as an FPGA accelerator.

engine selection process later in XLVM. The keyword, **fusion**, denotes a set of capabilities that can be sequentially executed internally in an engine, while avoiding external data communication with the host or other engines. For instance, line 21 illustrates that the fft capability can be fused with the band_pass_filter capability, while band_pass_filter cannot be fused with any other capabilities on DeCO. The **cost** construct lets engine developers specify a means to estimate the latency of capabilities. This can be mapped to a cycle-accurate simulator, hard-coded metric, or a machine learning-based cost models as in AutoTVM.

IV. Accelerator-Level Virtual Machine

The Yin-Yang abstractions need to be realized as a unified execution flow so that the application is executed efficiently and the maximal gains can be achieved from cross-domain multi-acceleration. To accomplish this objective, we devise Accelerator-Level Dataflow Virtual Machine (XLVM), which is at the confluence of the Yin-Yang abstractions. XLVM preserves and translates the CNF program as a queued-fractalized dataflow graph (QF-DFG) intermediate representation (IR). Then, we develop Engine Selector that selects the engines for the application, maximizing acceleration gains. Finally, we also develop Engine Compiler, which compiles the individual engines into the corresponding runnable executables and links them as a unified execution flow.

A. Queued-Fractalized Dataflow Graph (QF-DFG)

For effective engine selection and runtime orchestration, it is crucial to have an intermediate representation (IR) that 1) preserves the program and domain semantics (input, output, interface, and hierarchy of components), and 2) is flexible to support any granularity required for multi-acceleration. As such, we devise queued-fractalized dataflow graph (QF-DFG), which is designed to capture the details of the program such as dependency (order of execution), functionality (operation), and compositionality (hierarchy) of the CNF programs. In QF-DFG, each edge denotes a dataflow and node denotes an operation of multiple levels of granularity, progressing from coarse granular nodes to finer nodes until primitive scalar operations are reached. Figure 2(d) shows a snippet of the QF-DFG IR, which corresponds to the CNF program in Figure 2(b).

B. Engine Selector

QF-DFG is a target-independent IR which, when created from CNF, is oblivious to the target engines for execution, similar to target-independent IR stages of the traditional compilers. Unlike traditional compilation processes where the target platform is explicitly known, the duality of domains and engines provided by Yin-Yang opens a new avenue for optimal target engine determinations as it exhibits the following properties: 1) a domain possibly has multiple engines that can support different subsets of its capabilities; and 2) every engine, even within the same domain, has different performance energy tradeoffs. Thus, to choose an optimal combination of engines for a given QF-DFG and a set of engine specifications, XLVM is equipped with Engine Selector, which exploits 1) a simple cost model as a proxy to estimate the performance of engine assignment, and 2) an optimized objective function.

Algorithm 1

Engine selection algorithm for QF-DFG

```
1:  **Input:** QF-DFG $G(N,E)$
2:  **Output:** Engine Selection $S$
3:  candidates ← Ø, cost ← {}
4:  **while** new_candidate_exists() **do**
5:      candidates ← candidates ∪ {(n, e) | ∀ n ∈ $N$,∃ e ∈ n.domain.engines}
6:  **end while**
7:  **for** c **in** candidates **do**
8:      cost[c] ← 0
9:      **for** (n, e) **in** c **do**
10:         $\text{cost[c]} \leftarrow \text{cost[c]} + \mathscr{T}(\text{n, e})$
11:         **for** (n_child, e_child) **in** children((n, e)) **do**
12:             $\text{cost[c]} \leftarrow \text{cost[c]} + \mathscr{C}(\text{e, e\_child}) + \mathscr{D}(\text{e,e\_child})$
13:         **end for**
14:     **end for**
15: **end for**
16: $S$ ← find_engine_selection_with_minimum_cost(cost)
17: **return** $S$
```

Cost model and objective function.—We model the execution time of end-to-end multi-acceleration applications using three cost functions: 1) computation latency ($\mathscr{T}$), 2) data copy overhead ($\mathscr{C}$), and 3) data format conversion cost ($\mathscr{D}$). To simplify the design, we model the overall cost for the given QF-DFG as a sum of these functions, which does not consider the dynamic runtime factors such as pipelined execution and bandwidth contention with other applications. Using this cost model, we formulate the objective function of Engine Selector as a combination of engines S for the QF-DFG from the candidate engine set E, that minimizes total execution latency:

$$\underset{S \subset E}{\text{argmin}}\ \text{Cost} = \sum_{i} \mathscr{T}_i + \sum_{ij} \mathscr{C}_{ij} + \sum_{ij} \mathscr{D}_{ij}, \quad \text{for } i, j \in S$$

Algorithm 1 illustrates the engine selection process, which takes a QF-DFG and maps the graph nodes to a set of available engines that minimize the expected latency by optimizing the cost function. Engine Selector conducts a brute-force search of all possible engine assignment combinations to the QF-DFG and formulates the candidate set for the engine selections (Line 4–6). Then, Engine Selector evaluates the cost function per each candidate selection and chooses the candidate that imposes the minimum latency cost (Line 7–16). The selection results are augmented to the QF-DFG as metadata. While we demonstrated the engine selection process optimizing for the execution time, the objective function can be updated for other objectives such as energy efficiency or SLO.

C. Engine Compiler

Once every node has been assigned to an engine, Engine Compiler individually invokes the engine-specific compiler to obtain the engine executable. The canonical set of operations in engine executables constitutes loading the input data to engines, setting the configuration registers, triggering the computation, observing the runtime status, and receiving the output data. The underlying implementations of these operations for accelerators are all disparate, which makes the runtime orchestration difficult. To unify the interfaces, XLVM abstracts the engines as files that can perform computation and formalizes the engine interfaces as a set of file management APIs. Similar to the Unix I/O, these APIs include (1) **open** a new engine, (2) **read** data back from the engine, (3) **write** data to the engine, (4) initiate **compute** of a capability, and (5) **close** the engine. Thus, to link this *computational* file abstraction with the low-level hardware interfaces, the engine developers are asked to provide engine-specific device drivers.

V. Evaluation

A. Experimental Setup

Benchmarks.—Cross-domain multi-acceleration is an emerging field and there is a lack of established workloads that span multiple domains. We take real-life applications comprising well known algorithms to create a benchmark suite that can evaluate cross-domain multi-acceleration. Table I(a) summarizes these benchmarks, the domains they contain, and the accelerated kernels: (1) memory-enhance is the deep brain stimulation introduced in Section II-B; (2) robot-explorer is a four-wheeled robot equipped with a Kinect sensor to find its way through a cave and requires a KinectFusion (KF) algorithm to reconstruct a 3D map of the cave and MPC algorithm to navigate through the cave; (3) video-sync synchronizes subtitles with speech segments for video files, and requires MPEG-decoding and FFT to boost the speech-text pattern matching process; (4) stock-market predicts the call option price in the stock market, and requires sentiment analysis using LR on news articles to extract market signals with Black-Scholes model to predict call option pricing; (5) leukocyte detects leukocytes from video microscopy of blood vessels, and uses Gradient Inverse Coefficient of Variation (GICOV) score to perform detection in the frame and Motion Gradient Vector Flow (MGVF) matrix to track the leukocytes; (6) security-camera detects suspicious objects from its input video stream by decoding the MPEG encoded video stream using MPEG-Decoding and performing an object detection using deep learning (Tiny-Yolo-v2).

Compute platforms.—Table I(b) summarizes the domains used in the benchmarks, the accelerated capabilities, the used engines, and their platforms. Our system is equipped with a host CPU, Intel Xeon E3 (3.50 GHz). For fair comparison, we use optimized software libraries to obtain the best performance, including Intel MKL 2020, OpenBlas v0.3, and OpenCV 3.4.2. For FPGA, we use Xilinx KCU1500 with open-source hardware accelerators [7, 9]. Accelerators are attached to the host via a PCIe interconnect.

Runtime measurements.—We run the experiments for ten times and attain the average to report. When open-source RTL implementations of existing FPGA accelerators are unavailable, we use the author-provided simulators to measure the performance. Using the kernel execution time on the platforms, we estimate the end-to-end application runtime.

Energy measurements.—To measure the energy consumption, we use the Intel Running Average Power Limit (RAPL) for CPU and use the simulators for the FPGA accelerators.

Programming effort.—Accurately measuring human effort is impossible, yet similar to prior works, FlexJava and EnerJ, we count the number of lines of code (LoC) and the number of annotations to quantify the human effort.

B. Experimental Results

1) Performance Improvement—Figure 3(a) shows the speedup gains as the number of accelerator engines increases compared to the CPU-only baseline. All the benchmarks provide benefits even from a single-engine acceleration, which yields a 12.0× speedup when the best-performing engine is used for accelerating the benchmarks. However, the results show that the speedup increases to 29.4× on average when leveraging more engines, which amounts to a 145% extra speedup. Thus, there is untapped potential in accelerating multiple kernels, which is unleashed by our dual abstractions and XLVM. The rightmost bar (“Manual Program”) also shows the speedup when the maximal number of accelerators are enabled manually by programmers, which represent the ideal speedup that Yin-Yang would be able to achieve. The results show that Yin-Yang almost reaches this ideal speedup, while requiring less programming effort (see Section V-B(3)). Overall, our system attests to the common wisdom that “the more accelerators, the better”.

2) Performance-per-Joule Improvement—Figure 3(b) illustrates the performance-per-Joule improvement of multi-acceleration over the CPU baseline. As the figure shows, acceleration using a single engine achieves an overall Performance-per-Joule improvement of 7.0× against the baseline. By leveraging more engines through the Yin-Yang abstractions, we can achieve higher performance-per-Joule improvements of 12.2×, which is translated to 74.2% extra efficiency. Similar to Section V-B(1), we also report the manual multi-acceleration result, which shows that Yin-Yang closely reaches to the ideal efficiency gain, only leaving a marginal room for improvement.

3) Programmability—Figure 3(c) shows Lines of Code (LoC) improvements of the dual abstractions when compared to manual programming. The bar on the left represents LoC that programmers write, while the stacked bar on the right delineates the summated LoC that programmers, domain experts, and engine developers should write in aggregate. The results show that Yin-Yang effectively reduces the LoC by 33.1% on average while obtaining the same functionality and performance. The human efforts needed for domain descriptions and engine specifications is imposed only once when registering the domains and engines. From the programmers perspective, the LoC is reduced from 383 to 137, which increases the reduction rate to 64.2%. These results suggest that the proposed dual abstractions allow domain experts and engine developers to take part in enabling multi-acceleration with minimum effort, and CNF emancipates application programmers from the onerous task of hardware development and low-level programming for orchestrating multiple accelerators.

VI. Related Work

Abstractions for heterogenous platforms.

Although various general purpose abstractions for accelerators such as OpenCL exist, they do not incorporate the algorithmic domain knowledge. Intel oneAPI [10] provides libraries and compilation tools that can target multiple accelerators. The libraries of oneAPI contain fine-grained constructs that allow programmers to focus on their domain of interest and optimize it. SysML [11] is a system architecture modeling tool built upon Unified Modeling Language (UML). While SysML has a similarity to our CNF programming model, it offers a general-purpose and unified abstraction that lacks the notion of domains. In contrast, this work provides *cross-domain* programming abstractions and necessary mechanisms to make it easy for programmers to harmoniously combine existing accelerators from different domains together to develop a single application.

Domain-specific abstractions.

There are a plethora of one-sided acceleration solutions [6, 8] for a single domain, which is either *algorithm-centric* or *hardware-centric*. Our approach differs from these works in providing dual abstractions that move away from one-sided representation of a single domain and links multiple domains. This enables us to utilize disjointly pre-designed accelerators to be used in tandem for cross-domain applications.

FPGA acceleration.

High-Level Synthesis (HLS) is an effective tool that allows programmers to use a high-level language for accelerator development. While HLS improves programmability, its performance gains are usually lower than the custom-designed accelerators, as shown in prior works [12]. In contrary, Yin-Yang is an alternative programming tool that offers three different abstractions for three parties–(1) domain experts, (2) engine developers, and (3) application programmers, which allow them to collaboratively enable multi-acceleration for cross-domain applications.

VII. Conclusion

Cross-domain multi-acceleration can unlock new capabilities. For this emerging direction, we uniquely provide dual abstractions which preserve domain knowledge while linking algorithmic representations to hardware capabilities. As a mechanism to provide this linkage, we develop XLVM, which represents the program as QF-DFG and determines efficient engine-to-capability mappings. Experimental results using a real-life benchmark suite show significant improvements in performance and energy when multiple accelerators from different domain are used. This paper also provides an open-source benchmark suite for the emerging area of cross-domain multi-acceleration, which is available at https://github.com/he-actlab/cross-domain-benchmarks.

Acknowledgment

This work was in part supported by generous gifts from Google, Samsung, Qualcomm, Microsoft, Xilinx, as well as the National Science Foundation (NSF) awards CCF#2107598, CNS#1822273, National Institute of Health

(NIH) award #R01EB028350, Defense Advanced Research Project Agency (DARPA) under agreement number #HR0011–18-C-0020, and Semiconductor Research Corporation (SRC) award #2021-AH-3039. This work was also partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (No.2018–0-00503, Researches on next generation memory-centric computing system architecture; and No.2022–0-01037, Development of high performance processing-in-memory technology based on DRAM). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Qualcomm, Microsoft, Xilinx, Samsung, NSF, SRC, NIH, DARPA, the U.S. Government, and the Korean Government.

Biographies

Joon Kyung Kim is a software engineer at Apple. He was a research assistant at the University of California, San Diego. His research interests include operating systems, computer networks, and computer architecture. Kim received his MS in computer science from the Georgia Institute of Technology. Contact him at jkkim@eng.ucsd.edu.

Byung Hoon Ahn is a PhD candidate in computer science at the University of California, San Diego. His research interests include machine learning, compilers, and computer architecture. Ahn received his MS in computer science and engineering from the University of California, San Diego. Contact him at bhahn@eng.ucsd.edu.

Sean Kinzer is a PhD candidate at the University of California, San Diego. His research interests include compilers and domain-specific architectures. Kinzer received his MS in computer science and engineering from the University of California, San Diego. Contact him at skinzer@eng.ucsd.edu.

Soroush Ghodrati is a PhD candidate at the University of California, San Diego. His research interests include computer architecture and next-generation specialized systems for AI/ML. Ghodrati received his MS in computer science and engineering from the University of California, San Diego. Contact him at soghodra@ucsd.edu.

Rohan Mahapatra is a PhD student at UC San Diego. His research interest include designing systems for datacenters, serverless computing, and GPUs. Mahapatra received his MS in electrical and computer engineering from the University of Wisconsin-Madison. Contact him at rohan@ucsd.edu.

Brahmendra Yatham is a senior power architect at Nvidia. His research interests include machine learning hardware and computer architecture. Yatham received his MS in computer engineering from the University of California, San Diego. Contact him at byatham@nvidia.com

Shu-Ting Wang is a PhD student at UC San Diego. His research interests include designing systems and networking for datacenters, and serverless computing. Wang received his MS in computer science from the National Tsing Hua University. Contact him at shutingwang@ucsd.edu.

Dohee Kim is a software engineer at Viva Republica (TossBank). Her research interests include computer architecture and systems optimization. Kim received her MS in computer science from the Korea Advanced Institute of Science and Technology (KAIST). Contact her at dohee.kim@toss.im.

Parisa Sarikhani is a PhD student at Emory University. Her research interests include developing precision neuromodulation therapies using AI, and designing automated closed-loop neuromodulation frameworks using machine learning. Sarikhani received her MS in electrical engineering from Shiraz University. Contact her at psarikh@emory.edu.

Babak Mahmoudi is an assistant professor of Biomedical Informatics and Biomedical Engineering at Emory University. His research is broadly focused on developing intelligent systems for precision therapies and diagnostics using machine learning and artificial intelligence. Prior to joining the faculty at Emory, he completed an NIH fellowship in translation neurology. Mahmoudi received his PhD in biomedical engineering from the University of Florida. Contact him at b.mahmoudi@emory.edu.

Divya Mahajan is a senior researcher at Microsoft Azure's Cloud Accelerated Systems & Technologies team. Her research interests include devising next-generation sustainable compute platforms targeting end-to-end data pipeline for large scale AI and machine learning. Mahajan received her PhD in computer science from the Georgia Institute of Technology. Contact her at divya.mahajan@microsoft.com.

Jongse Park is an assistant professor in the School of Computing at the Korea Advanced Institute of Science and Technology (KAIST). He is also co-leading the Computer Architecture and Systems Laboratory (CASYS). His research focuses on building hardware-software co-designed systems for emerging algorithms and applications. Park received his PhD in computer science from the Georgia Institute of Technology. Contact him at jspark@casys.kaist.ac.kr.

Hadi Esmaeilzadeh is the Halicioğlu Chair in Computer Architecture at the University of California, San Diego where he is an associate professor of Computer Science & Engineering. He is the founder and CTO of Protopia AI. His research interests include computer architecture, machine learning, systems, and philosophy and psychology of conciseness. Esmaeilzadeh received his PhD in computer science and engineering from the University of Washington. He was inducted to the ISCA Hall of Fame in 2018, and he is the recipient of IEEE TCCA "Young Architect" Award and the Air Force Young Investigator Award. Contact him at hadi@eng.ucsd.edu.

References

[1]. "Aws f1," https://aws.amazon.com/ec2/instance-types/f1/.

[2]. Fowers J, Ovtcharov K, Papamichael M, Massengill T, Liu M, Lo D, Alkalay S, Haselman M, Adams L, Ghandi M et al., "A configurable cloud-scale dnn processor for real-time ai," in ISCA, 2018.

[3]. Liniger A, Domahidi A, and Morari M, "Optimization-based autonomous racing of 1: 43 scale rc cars," Optimal Control Applications and Methods, vol. 36, no. 5, pp. 628–647, 2015.

[4]. Künhe F, Gomes J, and Fetter W, “Mobile robot trajectory tracking using model predictive control,” in II IEEE latin-american robotics symposium, vol. 51. Citeseer, 2005.

[5]. Sarikhani P, Miocinovic S, and Mahmoudi B, “Towards automated patient-specific optimization of deep brain stimulation for movement disorders,” in EMBC, 2019.

[6]. Jain AK, Li X, Singhai P, Maskell DL, and Fahmy SA, “Deco: A dsp block based fpga accelerator overlay with low overhead interconnect,” in FCCM, 2016.

[7]. Mahajan D, Park J, Amaro E, Sharma H, Yazdanbakhsh A, Kim JK, and Esmaeilzadeh H, “TABLA: A unified template-based framework for accelerating statistical machine learning,” in HPCA, 2016.

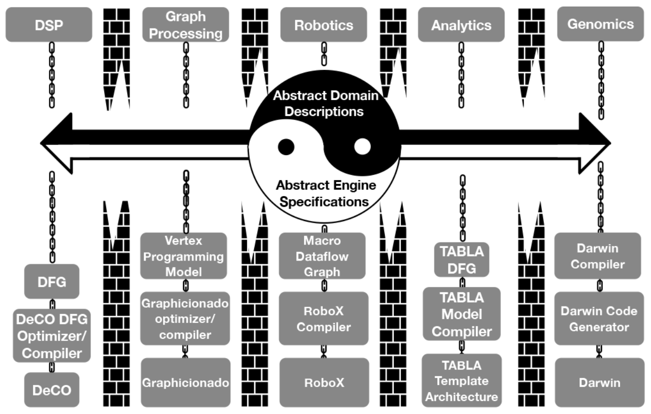
[8]. Sacks J, Mahajan D, Lawson RC, and Esmaeilzadeh H, “Robox: An end-to-end solution to accelerate autonomous control in robotics,” in ISCA, 2018.

[9]. Nardi L, Bodin B, Zia MZ, Mawer J, Nisbet A, Kelly PHJ, Davison AJ, Luján M, O’Boyle MFP, Riley G, Topham N, and Furber S, “Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM,” in ICRA, 2015.

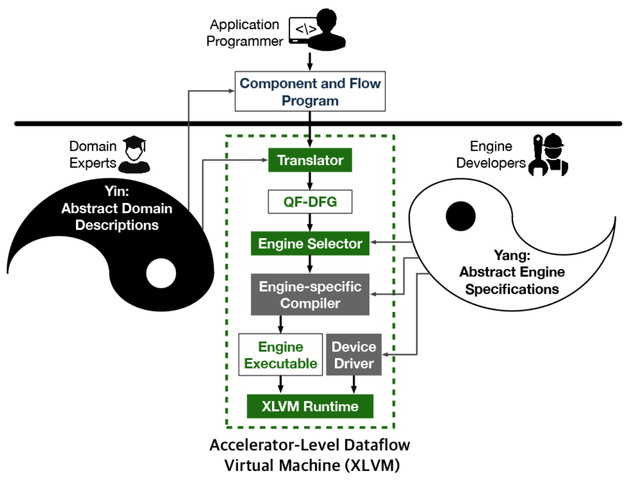
[10]. “Intel oneAPI,” https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[11]. Friedenthal S, Moore A, and Steiner R, A Practical Guide to SysML: The Systems Modeling Language, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[12]. Chang S-E, Li Y, Sun M, Shi R, So HK-H, Qian X, Wang1 Y, and Lin X, “Mix and Match: A Novel FPGA-Centric Deep Neural Network Quantization Framework,” in HPCA, 2021.

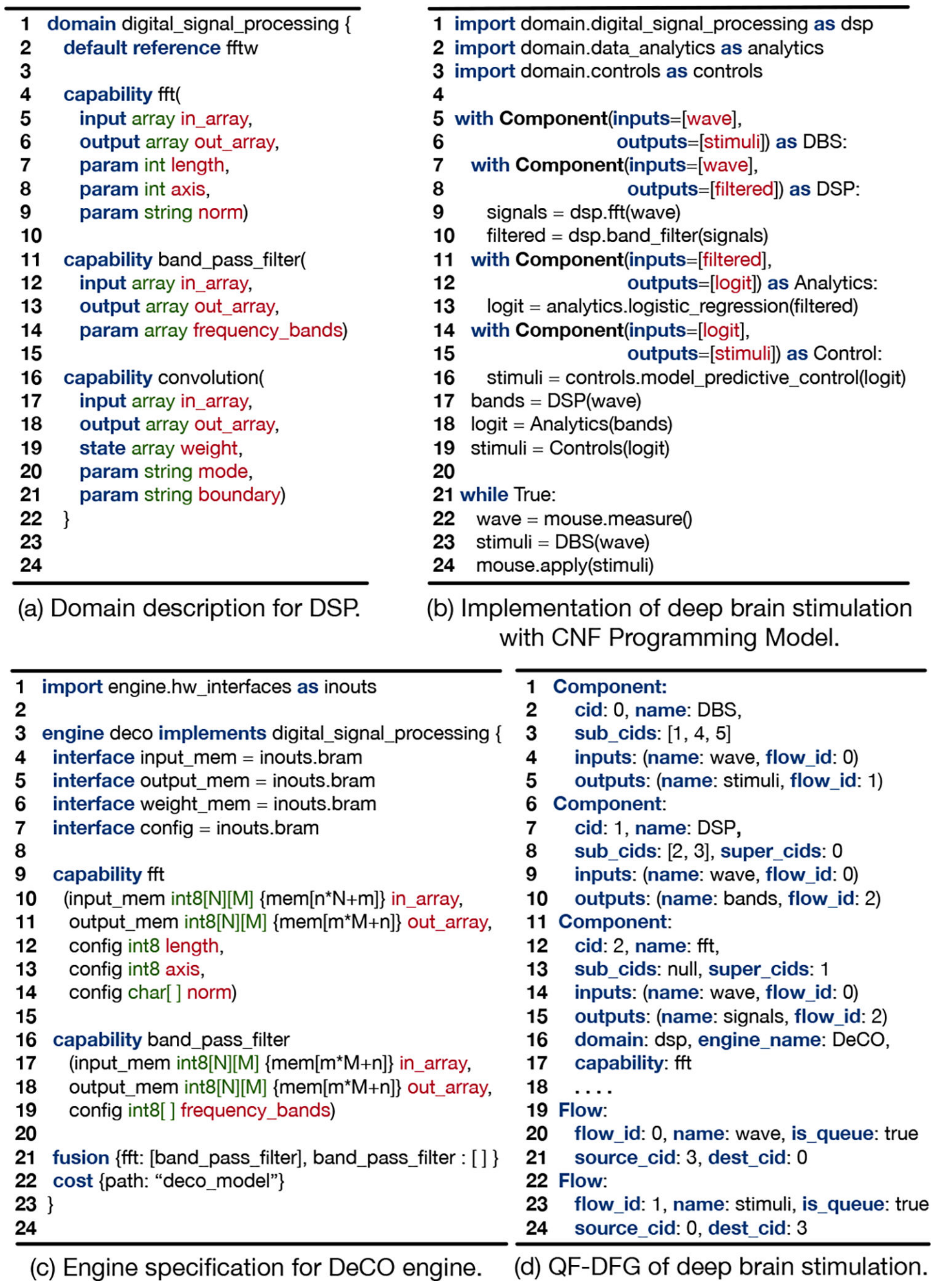


(a) Yin-Yang dual abstractions break the vertical barriers of domain-specific stacks and enable cross-domain multi-acceleration in the heterogeneous cloud.



(b) Yin-Yang dual abstractions and XLVM for cross-domain multi-acceleration.

Fig. 1:
Overview of Yin-Yang.



```
domain digital_signal_processing {
  default reference fftw

  capability fft(
    input array in_array,
    output array out_array,
    param int length,
    param int axis,
    param string norm)

  capability band_pass_filter(
    input array in_array,
    output array out_array,
    param array frequency_bands)

  capability convolution(
    input array in_array,
    output array out_array,
    state array weight,
    param string mode,
    param string boundary)
}
```

(a) Domain description for DSP.

```
import domain.digital_signal_processing as dsp
import domain.data_analytics as analytics
import domain.controls as controls

with Component(inputs=[wave],
               outputs=[stimuli]) as DBS:
  with Component(inputs=[wave],
                 outputs=[filtered]) as DSP:
    signals = dsp.fft(wave)
    filtered = dsp.band_filter(signals)
  with Component(inputs=[filtered],
                 outputs=[logit]) as Analytics:
    logit = analytics.logistic_regression(filtered)
  with Component(inputs=[logit],
                 outputs=[stimuli]) as Control:
    stimuli = controls.model_predictive_control(logit)
  bands = DSP(wave)
  logit = Analytics(bands)
  stimuli = Controls(logit)

while True:
  wave = mouse.measure()
  stimuli = DBS(wave)
  mouse.apply(stimuli)
```

(b) Implementation of deep brain stimulation with CNF Programming Model.

```
import engine.hw_interfaces as inouts

engine deco implements digital_signal_processing {
  interface input_mem = inouts.bram
  interface output_mem = inouts.bram
  interface weight_mem = inouts.bram
  interface config = inouts.bram

  capability fft
   (input_mem int8[N][M] {mem[n*N+m]} in_array,
    output_mem int8[N][M] {mem[m*M+n]} out_array,
    config int8 length,
    config int8 axis,
    config char[ ] norm)

  capability band_pass_filter
    (input_mem int8[N][M] {mem[m*M+n]} in_array,
    output_mem int8[N][M] {mem[m*M+n]} out_array,
    config int8[ ] frequency_bands)

  fusion {fft: [band_pass_filter], band_pass_filter : [ ] }
  cost {path: “deco_model”}
}
```

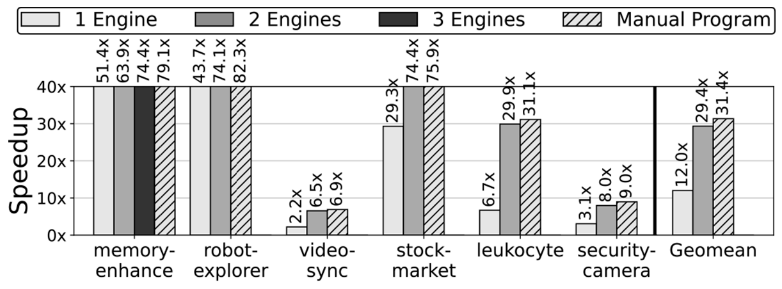
(c) Engine specification for DeCO engine.

```
Component:
  cid: 0, name: DBS,
  sub_cids: [1, 4, 5]
  inputs: (name: wave, flow_id: 0)
  outputs: (name: stimuli, flow_id: 1)
Component:
  cid: 1, name: DSP,
  sub_cids: [2, 3], super_cids: 0
  inputs: (name: wave, flow_id: 0)
  outputs: (name: bands, flow_id: 2)
Component:
  cid: 2, name: fft,
  sub_cids: null, super_cids: 1
  inputs: (name: wave, flow_id: 0)
  outputs: (name: signals, flow_id: 2)
  domain: dsp, engine_name: DeCO,
  capability: fft
  . . . .
Flow:
  flow_id: 0, name: wave, is_queue: true
  source_cid: 3, dest_cid: 0
Flow:
  flow_id: 1, name: stimuli, is_queue: true
  source_cid: 0, dest_cid: 3
```

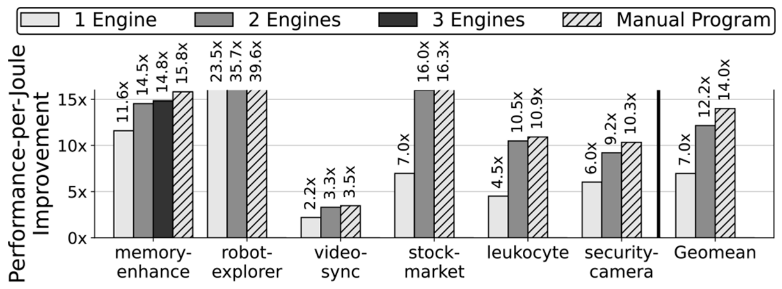
(d) QF-DFG of deep brain stimulation.

Fig. 2:

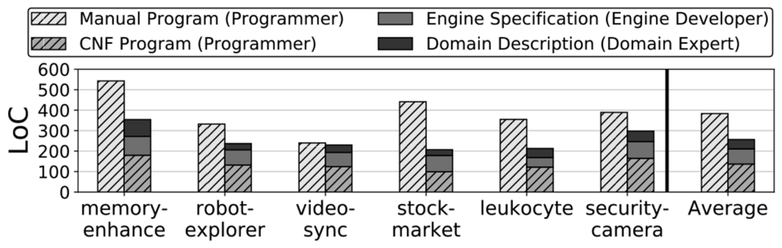
Example Yin-Yang code and its QF-DFG IR.



(a) Speedup with various number of accelerator engines against CPU baseline.



(b) Performance-per-Joule improvement achieved by multi-acceleration.



(c) Lines of Code (LoC) improvements of Yin-Yang in comparison with the baseline manual programming.

Fig. 3:

Experimental results that show the acceleration gains by Yin-Yang, in comparison with the baselines. We use the CPU-only execution as the baseline for performance and energy-efficiency results. The baseline for Lines-of-Code results is the case of manual programming.

TABLE I:

(a) Cross-domain benchmark suite, and (b) domains and engines used in the benchmarks.

(a)

Name	Description	Domains	Used Capabilities
memory-enhance	Deep Brain Stimulation closed-loop control pipeline to optimize stimulation signals for memory enhancement	DSP	FFT
		Data Analytics	LR
		Optimized Control	MPC
robot-explorer	KinectFusion for 3D map generation with model predictive control for cave exploration	Robotics	KF
		Computer Vision	MPC
video-sync	Calculate correct offset to sync a movie and subtitle file	DSP	MPEG-Decode
			FFT
stock-market	Text sentiment classification on stock market news articles to estimate call option price	Data Analytics	LR
		Finance	Black-Scholes
leukocyte	Detect and tracks rolling leukocytes (white blood cells) in a microscopy of blood vessels vivo video	Computer Vision	GICOV
			MGVF
security-camera	Real-time object detection system which decodes MPEG encoded video	Deep Learning	Tiny-Yolo-v2
		DSP	MPEG-Decode

(b)

Domain	Capabilities	Engine *	Platform
DSP	FFT, MPEG-Decode	FFTW	CPU
		ffmpeg	CPU
		DeCO	FPGA
		LogiCore	FPGA
Data Analytics	LR	MLPack	CPU
		InAccel	FPGA
		Tabla	FPGA
Robotics	MPC	ACADO	CPU
		RoboX	FPGA
Computer Vision	GICOV, MGVF, KF	OpenCV	CPU
		SLAMBench	FPGA
		Iron	FPGA
Finance	Black-Scholes	QuantLib	CPU

		HyperStreams	FPGA
		TensorFlow	CPU
Deep Learning	Tiny-Yolo-v2	TVM	CPU
		DnnWeaver	FPGA

*We omit the references due to the number of reference limit for submission.