A Computational Stack for Cross-Domain Acceleration

Sean Kinzer Joon Kyung Kim Soroush Ghodrati Brahmendra Yatham
Alric Althoff* Divya Mahajan† Sorin Lerner Hadi Esmaeilzadeh

Alternative **C**omputing **T**echnologies (**ACT**) Lab
University of California San Diego * Tortuga Logic † Microsoft

{skinzer,jkkim,soghodra,byatham}@eng.ucsd.edu alric@tortugalogic.com
divya.mahajan@microsoft.com lerner@eng.ucsd.edu hadi@eng.ucsd.edu

***Abstract*—Domain-specific accelerators obtain performance benefits by restricting their algorithmic domain. These accelerators utilize specialized languages constrained to particular hardware, thus trading off expressiveness for high performance. The pendulum has swung from one hardware for all domains (general-purpose processors) to one hardware per individual domain. The middle-ground on this spectrum–which provides a unified computational stack across multiple, but not all, domains–is an emerging and open research challenge. This paper sets out to explore this region and its associated tradeoff between expressiveness and performance by defining a cross-domain stack, dubbed PolyMath. This stack defines a high-level *cross-domain language (CDL)*, called PMLang, that in a modular and reusable manner encapsulates mathematical properties to be expressive across multiple domains–Robotics, Graph Analytics, Digital Signal Processing, Deep Learning, and Data Analytics. PMLang is backed by a *recursively-defined* intermediate representation allowing *simultaneous access* to all levels of operation granularity, called *sr*DFG. Accelerator-specific or domain-specific IRs commonly capture operations in the granularity that best fits a set of Domain-Specific Architectures (DSAs). In contrast, the recursive nature of the *sr*DFG enables simultaneous access to all the granularities of computation for every operation, thus forming an ideal bridge for converting to various DSA-specific IRs across multiple domains. Our stack unlocks multi-acceleration for end-to-end applications that cross the boundary of multiple domains each comprising different data and compute patterns.**

Evaluations show that by using PolyMath it is possible to harness accelerators across the five domains to realize an average speedup of 3.3× over a Xeon CPU along with 18.1× reduction in energy. In comparison to Jetson Xavier and Titan XP, cross-domain acceleration offers 1.7× and 7.2× improvement in performance-per-watt, respectively. We measure the cross-domain expressiveness and performance tradeoff by comparing each benchmark against its hand-optimized implementation to achieve 83.9% and 76.8% of the optimal performance for single-domain algorithms and end-to-end applications. For the two case studies of end-to-end applications (comprising algorithms from multiple domains), results show that accelerating all kernels offers an additional 2.0× speedup over CPU, 6.1× improvement in performance-per-watt over Titan Xp, and 2.8× speedup over Jetson Xavier compared to only the one most effective single-domain kernel being accelerated. Finally, we examine the utility and expressiveness of PolyMath through a user study, which shows, on average, PolyMath requires 1.9× less time to implement algorithms from two different domains with 2.5× fewer lines of code relative to Python.

***Index Terms*—Accelerator design, Machine learning systems, Algorithms/System Co-design, Domain Specific Language**

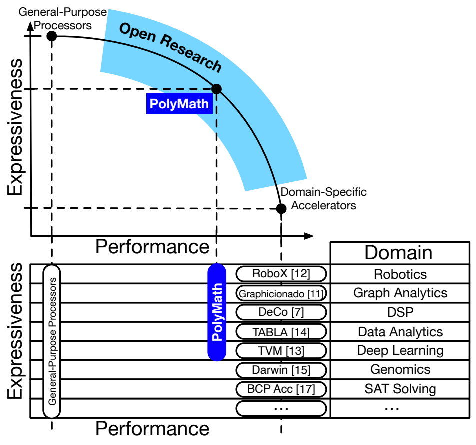


Fig. 1: Emerging tradeoff: Expressiveness vs. Performance.

I. Introduction

End-to-end applications ranging from delivery drones [1] to smart speakers [2] cross multiple domains. One such application senses the environment, (1) pre-processes the sensory data, feeds it to a (2) perception module that in turn invokes a (3) decision making process to determine actions. Perception is currently reigned by Machine Learning (ML), which has attracted significant attention, but applications are not just ML. Sensory data processing relies on algorithms from Digital Signal Processing (DSP) while Control Theory and Robotics bring forth the final action that may also feed the perception module. Even though these domains work in tandem to realize an entire application, they are becoming isolated by the current push towards Domain-Specific Accelerators (DSAs). One the one hand, these accelerators tradeoff generality for performance and energy efficiency by restricting programmability to a single domain [3]. On the other hand, the traditional general-purpose computational stack cannot meet the computational demands of emerging applications [4–6]. Although, Domain-Specific Accelerators (DSA) bridge this performance gap, but make implementation an arduous task of dealing with isolated programming interfaces. Thus, expressiveness is becoming limited, making the composition of an end-to-end application a major challenge for execution on accelerators. As a consequence, users seeking to create compute-intensive applications composed of algorithms from different domains must choose between either using a lower-performance, general-purpose processor or bear the burden of manually stitching together various domain-specific accelerators.

This emerging challenge creates a new tradeoff between performance and expressiveness, illustrated in Figure 1. On one extreme, we have General-Purpose Processors that allow expressing *all* domains at the cost of performance and/or efficiency [3]. On other extreme, are domain-specific accelerators [7–13] that can only support a *single* domain to be executed on one particular specialized architecture, thus are very performant. Even though certain DSAs offer computational stacks, composing an end-to-end application that crosses the boundary of many domains requires intimate knowledge of multiple different interfaces and various hardware accelerators to obtain high performance. Recent efforts such as, Graphicionado [14], RoboX [15], TVM [16], Tabla [17], aim to unify high-level coding within a single domain, cross-domain stacks for accelerators still remains an open challenge (Figure 1).

As Figure 1 illustrates, by addressing this challenge, PolyMath defines a new point in the Expressiveness vs. Performance design space. The ingredients of our approach are:

1) Exploiting the mathematical similarities across domains to design a modular and reusable language, PMLang, that is expressive across Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning. It offers one-to-one mapping between code and mathematical formulation while retaining modularity, thus making it familiar to both domain-experts and software engineers. PMLang offers lightweight type modifiers based on domain semantics to enable accelerators to handle on-chip and off-chip data allocation, storage, and transfer. PMLang is not an abstraction over existing domain specific languages, rather it explores a novel dimension of designing a unified, stand-alone language across multiple domains. This unique dimension defines a class of languages that we refer to as Cross Domain Languages (CDLs). To enable cross-domain acceleration, PMLang and its associated compilation stack takes an initial step to bridge CDLs with domain-specific architectures, that are constrained to a single domain.
2) To preserve expressiveness and provide flexibility for compilation to different accelerators, we devise an intermediate representation that is a *recursively-defined* Dataflow Graph, providing *simultaneous* access to all levels of operation granularity (srDFG). The compute granularity of the kernels is not uniform across different accelerators, required for cross-domain settings. Thus, we define srDFG recursively in that its nodes are also srDFGs. As such, srDFG uniquely offers simultaneous access to various levels of computation granularity within a single program, thus enabling leveraging different accelerators. This capability enables cross-domain multi-acceleration–acceleration of a cross-domain application on different accelerators. The flexible, recursive nature of our IR shown in Figure 2, whose edges preserve the type modifier metadata.
3) PolyMath uses a modular compilation framework that conveniently enables creation and application of pipelined compilation passes on the srDFG. To convert srDFGs to executable accelerator code, PolyMath offers a graph lowering algorithm with a conversion strategy which uses metadata embedded in the srDFG edges to flexibly translate the graph nodes. The lowering algorithm applies transformations which produce a new srDFG made up of compute kernels at the same granularity of the target accelerator. Once lowered, the metadata associated with the srDFG edges is translated to the accelerator's own IR for final binary generation through its own scheduling and mapping framework.
4) Last but not least, PolyMath will be the very first extensible, modular, and open-source computation stack to enable the community to innovate and explore the impending challenges of cross-domain acceleration at a time when domain-specific compilation stacks, except for DNNs, are elusive. We have made the entire codebase available in a public repository (https://github.com/he-actlab/polymath). Although there are many accelerators in the literature, many of their compilation stacks are not available. By making PolyMath open-source and extensible, the community can add other domains which align with the core mathematical constructs in PMLang.

We show PolyMath's balance of expressiveness and performance by compiling twelve different algorithms across robotics, graph analytics, digital signal processing, data analytics, and deep learning. These workloads achieve an overall speedup of 3.3× over a Xeon CPU along with 18.1× reduction in energy. In comparison to Titan Xp and Jetson Xavier GPUs, cross-domain acceleration offers 7.2× and 1.7× in energy reduction. We next measure the tradeoff of cross-domain algorithm expression and find that PolyMath can achieve 83.9% of the performance of the same algorithms implemented in their native stack's language. We also study two end-to-end applications that cross multiple domains. Accelerating all the kernels offers an additional 2.0× speedup over CPU, 6.1× additional improvement in energy requirements over Titan Xp, and 2.8× speedup over Jetson Xavier vs when only one most effective kernel was accelerated. The results show that when only a part is accelerated, the slower non-accelerated kernels dictates the overall improvement, whereas, when all the algorithms in the application are accelerated Amdahl's burden reduces, and the improvement of all the domains is magnified. To evaluate the usability and expressiveness of PMLang relative to Python, we conduct a user study and found that on average PMLang required 1.9× less time to implement algorithms with 2.5× fewer lines of code. Finally, end-to-end performance of Polymath is 76.8% of the performance of two manually implemented end-to-end applications. Given the fact that PolyMath offers greater ease of programming compared to Python, the automation overhead of 23.1% (=100%-76.8%) is a fair bargain.

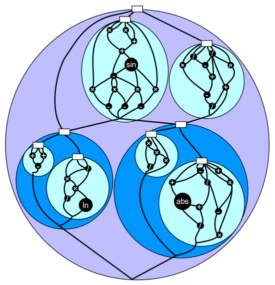


Fig. 2: Visual representation of PolyMath's srDFG.

II. PMLang: Mathematical Programming Interface

PMLang is designed to encapsulate the mathematical properties of these domains, as they are tied together by similar operations on multi-dimensional data, include minimal control-flow, and share use-cases such as cyber-physical systems. Consider the following application in its entirety: An end-to-end neuroscience application requires multiple domains to study the impact of deep brain stimulation on movement disorders and goes through the following steps: (1) convert raw electrocorticographic (ECoG) brain signals to frequency domain using fast Fourier transform (FFT); (2) apply logistic regression to classify these frequency domain signals into various biomarkers; (3) based on the classification, use model predictive control to send an optical stimulation back to the brain. This application crosses three domains, DSP, Data Analytics, and Control Theory in each iteration to generate deep brain stimulation signals.

There are numerous domain specific architectures for each of these algorithms/domains individually; however, using them for this application would require writing each part of the application in a different DSL, compiling them separately, and manually joining their executables. Instead, PMLang allows users to write their application as a *single program*, thus, eliminating the overhead of stitching together stacks to execute the program across multiple domain specific architectures.

Keeping the properties of target domains in mind, PMLang is designed to reduce the time to code a mathematical expression into a formula-based textual format, enabled by language constructs for modularity and light-weight type modifiers. Moreover for code organization and reduction in implementation time, PMLang includes reusable execution code blocks called components that perform operations on flows of data. These components encapsulate a task comprised of either other components and/or mathematical expressions which use traditional, imperative syntax to facilitate familiarity for experienced programmers. For *modularity* and *reusability*, components have distinct boundaries and arguments which are distinguished by type modifiers consisting of input, output, state, and param; each of which is associated with how the component will use the argument, shown in Table I. By using type modifiers in component arguments to explicitly identify data semantics PMLang binds operations to data being operated on for accelerators to take advantage of.

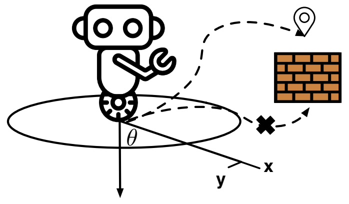


Fig. 3: Visual of trajectory tracking for wheeled robot.

The remainder of the section will delve into details of PMLang constructs through an example. For brevity, we show the PMLang program for Model Predictive Control (MPC) from Control Theory used for Robotics. MPC attempts to solve a constrained optimization problem over a finite sequence of inputs. MPC can also be used for the aforementioned brain stimulation application. We provide MPC in the context of a mobile two-wheeled robot performing trajectory tracking, shown in Figure 3. Here, the sensors send the current state of the robot as inputs to a model that predicts the next location, then optimizes a sequence of control signals for moving the robot to the predicted location. The program finds the optimal sequence of control signals, `ctrl_mdl`, over a finite period of time to match a reference trajectory, `pos_ref`, that specifies the position and orientation of the robot. At each point in time, the actual position and orientation of the robot is input to the algorithm, which optimizes the `ctrl_mdl` and sends the next control signal, `ctrl_sgnl`, back to the robot. This process is summed up in the following steps:

Input: `pos` $\rightarrow$ the (x,y,θ) orientation of the robot.
Output: `ctrl_sgnl`$\rightarrow$ (ν, ω) control consisting the velocity and angular velocity to be sent to the robot.
State: $u \rightarrow (v, \omega)$ linear and angular velocities across a predetermined time horizon `h`.
Step 1: Make a prediction Using input `pos` and cost matrices `P` and `H`, predict the position and angle of the robot across horizon `h`.
Step 2: Compute the gradient of the objective function Calculate the error on the predicted position and orientation, `pos_pred`, using pre-computed gradient coefficient matrices `HQ_g` and `R_g`.
Step 3: Update the control model and send the output signal Using gradient, `g`, update the control model, and send the output control signal, `ctrl_sgnl`.

A. Components

Components form the building blocks of PMLang, and are used to delineate different parts of the program into multiple levels of execution. To delineate the access semantics for each argument of the components, PMLang uses type modifiers ((input), `output`, `state`, `param`). Using type modifiers relieves programmers from concern about the underlying accelerator-specific mechanisms for data exchange between different components. An `input` argument is used to feed data into the component, and is read-only; an `output` argument is used to return data from a component, and can only be written to; a `state` argument can be read or written

```
1  predict_trajectory(input float pos[a],
2                     input float ctrl_mdl[b],
3                     param float P[c][a],
4                     param float H[c][b],
5                     output float pred[c]){
6    index i[0:a-1], j[0:b-1], k[0:c-1];
7    pred[k] = sum[i](P[k][i]*pos[i]);
8    pred[k] = pred[k] + sum[j](H[k][j]*ctrl_mdl[j]);
9  }
10 update_ctrl_model(input float ctrl_prev[b],
11                   input float g[b],
12                   output float ctrl_mdl[b],
13                   output float ctrl_sgnl[s],
14                   param int h){
15   index i[0:b-2], j[0:s-1];
16   ctrl_sgnl[j] = ctrl_mdl[h*j];
17   ctrl_mdl[(h-1)*j] = 0;
18   ctrl_mdl[i] = ctrl_prev[(i+1)*h] - g[(i+1)*h];
19 }
20 mvmul(input float A[m][n],
21       input float B[n],
22       output float C[m]){
23   index i[0:n-1], j[0:m-1];
24   C[j] = sum[i](A[j][i]*B[i]);
25 }
26 compute_ctrl_grad(input float pos_pred[c],
27                   input float ctrl_mdl[b],
28                   input float pos_ref[c],
29                   param float HQ_g[b][c], // Input Cost Gradient
30                   param float R_g[b][b], // Cost Inverse Hessian
31                   output float g[b]){
32   index i[0:b-1], j[0:c-1];
33   float P_g[b], H_g[b];
34   err[j] = pos_ref[j] - pos_pred[j];
35   mvmul(HQ_g, err, P_g);
36   mvmul(R_g, ctrl_mdl, H_g);
37   g[i] = P_g[i] + H_g[i];
38 }
39 main(input float pos[3],
40      state float ctrl_mdl[20],
41      param float pos_ref[30],
42      param float P[30][3],
43      param float HQ_g[20][30],
44      param float H[30][20],
45      param float R_g[20][20],
46      output float ctrl_sgnl[2]){
47   float pos_pred[30], g[20];
48   index i[0:9], j[0:1];
49 RBT: predict_trajectory(pos, ctrl_mdl, P, H, pos_pred);
50 RBT: compute_ctrl_grad(pos_pred,ctrl_mdl,pos_ref,HQ_g,R_g,g);
51 RBT: update_ctrl_model(ctrl_mdl, g, ctrl_mdl, ctrl_sgnl,10);
52 }
```

Fig. 4: MPC for MobileRobot trajectory tracking in PMLang.

to, and represents data that is part of the state of the component, thus is preserved across invocations/iterations; and a `param` argument is a constant that is used to parameterize the component. These type modifiers describe whether or not the data will be re-used (`state`), kept unchanged (`param`), or used once and discarded (`input`/`output`). As an example, line 1 shows that argument `pos` is an `input` to the `predict_trajectory` component. As another example, line 41 shows a `state` argument named `ctrl_mdl` which indicates that `ctrl_mdl` is used, updated, shared across invocations of `main`, which matches the MPC semantics of optimizing the control model over a series of time steps. Type modifiers also enable custom accelerators to place `input` data such as `pos` in Read-only FIFO buffers to reduce data communication overhead and hardware memory logic, or store `state` data such as `ctrl_mdl` on-chip on the accelerator for fast repeated data accesses. Using a single set of type modifiers to describe data semantics across multiple domains unifies program implementation for end-to-end applications. This is exemplified in robotics and deep learning, where one domain uses "model" and the other "weight" to describe the same data semantics, both of which are described as `state` data in PMLang.

In addition to being reusable, these components allow users to conceive their program as a collection of sub-steps at varying levels of granularity making it adaptable for compilation to different accelerators. To instantiate a component, the programmer specifies its name and arguments. An example of component instantiation is shown in line 49-51 where `predict_trajectory`, `compute_ctrl_grad`, and `update_ctrl_model` is instantiated. Each instantiation creates a copy of the component, as if it were inlined. This is in contrast to conventional languages that rely on a function call stack which is sequential in nature. Instead, inlining enables the program to be mapped to our srDFG IR, which preserves opportunities for parallelism based on data flow dependencies.

B. Index Variables

PMLang is based on mathematical notations that do not use for loops and instead use indices (e.g., $\sum_{i=0}^{n-1} A_i$). To simplify programming based on formulae, PMLang uses `index` variables to concisely specify operations performed over ranges of multi-dimensional data without using explicit for loops. In its most basic form, an `index` variable represents a range of integers, specified by its lower and upper bounds. Line 48 shows two such `index` variable declarations: `i` and `j`. This approach reveals the inherent parallelism in mathematical formulae since operations expressed using this approach are naturally vectorizable without performing any loop transformations. For example, below is a PMLang statement iterating over all `j`s, each of which can be performed in parallel.

```
index j[0:s-1];
err[j] = pos_ref[j] - pos_pred[j];
```

Strided indexing. To support strided/non-sequential indexing (e.g., convolution), PolyMath also supports arithmetic operations on index variables as shown below.

```
ctrl_mdl[i] = ctrl_prev[(i+1)*h] - g[(i+1)*h];
```

Boolean conditional over indexing. Unlike a domain-specific language such as TABLA [17] that focuses solely on data analytics, PMLang allows Boolean conditionals to be applied to indices, which provides support for other domains such as graph analytics and robotics. For instance, the following computes the sum of the non-diagonal parts of the matrix `A`:

```
index i[0:N-1], j[0:M-1];
res = sum[i][j: j != i](A[i][j]);
```

Support for Boolean conditionals and non-sequential `index` variables flexibly incorporates common as well as specific-to-domain characteristics of algorithms across robotics, graph analytics, DSP, data analytics, and deep learning. These features distinguish PMLang from DSLs which either use (1) concise operations expressed through a fixed API (e.g., named functions such as "dense" in TVM [16]), or (2) simply do not support these construction since the specific target domain does not require them (e.g., TABLA [17].)

C. Mathematical Operations

Index variables allow for a nearly one-to-one mapping between mathematical notation and PMLang code. PMLang offers standard mathematical operators to be used with multi-dimensional data, expressed in a single statement by using index variables. PMLang's syntax for math expression of $C_j = \sum_{i=0}^{n} A_{j,i} \times B_i$ is:

```
C[j] = sum[i](A[j][i]*B[i]);
```

Non-Linear operations. PMLang includes a set of built-in functions to be used in math expressions commonly used across the multiple target domains, including non-linear operations such as cosine/sine (DSP, robotics), gaussian (robotics, DSP, data analytics), sigmoid/ReLU (deep learning, data analytics), etc. Including non-linear operations as part of PMLang simplifies algorithm expression and allows PolyMath to leverage the performance benefits of non-linear compute units in custom accelerators.

Reduction operations. PMLang is also equipped with built-in group reduction operations such as `sum`, `prod`, `max` etc., to calculate the summation ($\sum$), product (Π), or maximum value of a sequence of numbers. These group reductions operations are converted to srDFG with two levels of granularity: (1) the outer group DFG node that (2) encapsulates the scalar inner operations (nodes). This multi-granular representation enables the compiler to map the the outer encompassing node to a dedicated unit if the accelerator harbors it. Otherwise, the inner basic nodes are mapped to individual ALUs. This crucial flexibility is a unique feature of PolyMath and enables it be cross domain and target different accelerators.

Custom reduction operations. PMLang also supports custom group reduction operations, as they are commonly used in graph analytics and DSP algorithms. Custom reduction operations can be defined in PolyMath by specifying the arithmetic for a given set of input arguments. Below is an example of the definition of the `min` reduction function and using it to find the minimum value for the matrix `A`:

```
reduction min(a,b) = a < b ? a : b;
res = min[i][j](A[i][j]);
```

D. *Domain Annotations*

PMLang uniquely targets multiple domains, each of which is eventually accelerated with a Domain-Specific Architecture. As such, PMLang offers a light-weight mechanism to specify the target domain for *only top-level* component instantiations without tying it to a specific accelerator. All of the code within a component also inherits the same domain, which alleviates the programmer from having to annotate all component instances in their program. This is done by simply adding one of the five keywords: `RBT` (Robotics), `GA` (Graph Analytics), `DSP` (Digital Signal Processing), `DA` (Data Analytics), and `DL` (Deep Learning), as demonstrated below:

```
 RBT: predict_trajectory(pos, ctrl_mdl, P, H, pos_pred);
```

III. Simultaneous-Recursive DataFlow Graph

Accelerator-specific or domain-specific IRs commonly capture operations in the granularity that best fits the target architecture. One of the major challenges that PolyMath faces is targeting multiple domains each of whose accelerators operate on different granularities of computation. Even within a single domain, various architectures accept computation in different granularities. To address this challenge, we designed srDFG, an intermediate representation which is a *recursively-defined* dataflow graph, and provides *simultaneous* access to each level of recursion, srDFG. Our srDFG enables the compiler to simultaneously access all the granularities of computation for every component, thus forming the ideal bridge to convert to various accelerator-specific IR. Furthermore, srDFG enable multi-acceleration for end-to-end applications that cross the boundary of multiple domains with different data and compute patterns across Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning. Next, we describe the srDFG structure using Figure 5, a visual representation of the MobileRobot algorithm described in Section II.

TABLE I: A subset of PMLang's keywords and definitions.

Language Construct	Keyword	Description
Component	`string name`	Takes input, produces output, and reads/writes to state arguments
Domain	`RBT, GA, DSP, DA, DL`	Specifies a component's target domain
Type Modifiers	`input`	Flow of data, can be exclusively read from within a component scope
	`output`	Flow of data, can be exclusively written to within a component scope
	`param`	Constant parameter used to parameterize a component
Index Types	`state`	Flow of data, can be written to or read from within a component scope
	`index`	Specifies ranges of operations
Types	`bin, int, float, str, complex`	Data types used to for variable declarations.

A. sr*DFG Definitions*

An srDFG is defined as a pair, *(N,E)*, of nodes N representing PMLang operations, and edges E representing input or output operands. An srDFG node $n \in N$ is a pair *(name, srdfg)* of a string representing the name of an operation, and its lower-granularity operations srDFG composition. Each numbered box in Figure 5 represents the *srdfg* for different nodes at varying granularities within the MobileRobot algorithm. As shown in ❷: both the the subtraction operation and the `mvmul` component are nodes. An edge $e \in E$ is a tuple of source src and destination dst nodes, and the edge metadata md: *(src,dst,md)*. Edge metadata consists of the type, type modifier, and shape of the operand associated with the edge. For math operations, input and output edges represent operands and results, whereas adjacent edges in component instantiations represent state, input and output arguments. This is illustrated in ❶, where `pos` is the input argument, `ctrl_sgnl` is the output argument, and `ctrl_mdl` is the state argument that creates a cycle for multiple iterations. Given a node n, we denote the name and *srdfg* as *n.name* and *n.srdfg*, and similarly denote *src,dst,md* in an edge e as *e.src*, *e.dst*, and *e.md*. Lastly, the domain annotations previously described are translated to the *srdfg.domain* attribute for each *srdfg*.

B. sr*DFG Semantics*

As an example, the srDFG shown in ❸ will begin operations when the data in edges `R_g` and `ctrl_mdl` are

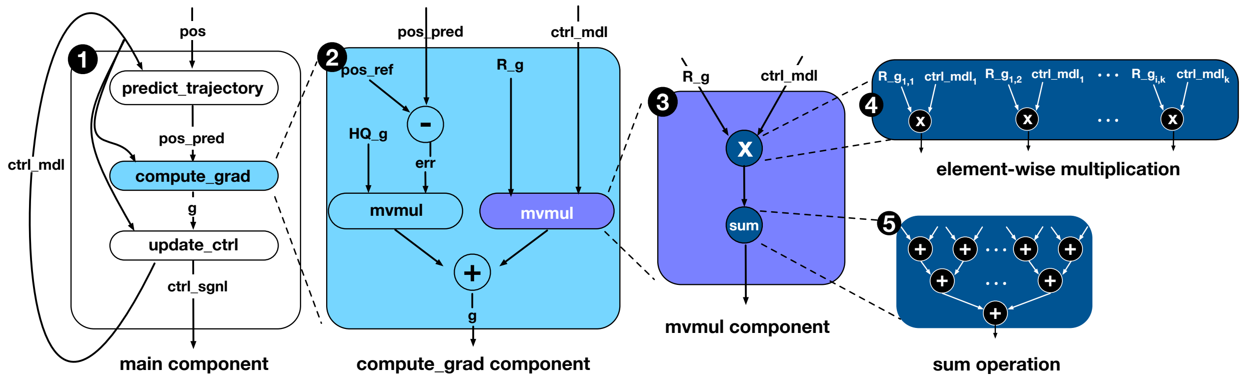


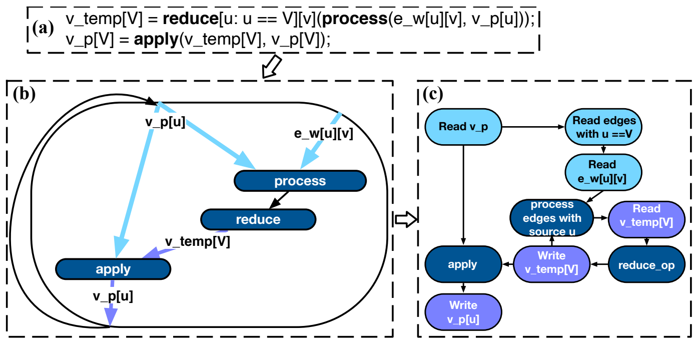
Fig. 5: Overview of the srDFG MobileRobot algorithm including zoomed-in views of its multiple levels of recursion.

ready. Each srDFG is a statically defined graph representing a single instantiation on its input values, with each component instantiation or operation getting its own srDFG, which allows for computing context sensitive information. As an example, Figure 5 ❷ shows two unique nodes and pairs of input edges for the `mvmul` component, and as a result each instantiation of `mvmul` gets its own node and srDFG. The srDFG in ❷ also shows how edges propagate their metadata to the lower granularity nodes, as the shapes of `R_g` and `ctrl_mdl` determine the number of element-wise multiplication nodes in ❹. The type modifier included in edge metadata can change depending on its srDFG, as shown in ❶ where the `ctrl_mdl` edge is a reusable *state*, but is an input edge for ❷.

C. Enabling Different Accelerators

Custom accelerators support unique set of operations performed on a variety of different typed and shaped inputs and outputs. To ensure flexibility, each srDFG includes operations as nodes, n, as well as the more fine-grained operations to define the node, *n.dfg*. To illustrate this point, each srDFG in Figure 5 represents a possible operation supported by an accelerator. If ❷ is supported by the target accelerator, the srDFG can be transformed to consist only of the operations represented by each node in ❶. If ❷ is not supported but ❸ is supported, then the operations in both ❶ and ❷ will be selected for compilation. PolyMath's base unit of lowering is a node, and if the nodes in the srDFG cannot be lowered to a specific hardware because of unsupported nodes, the compilation fails for that accelerator.

Each of these accelerator operations is closely tied to the types and shapes of its operands. The srDFG uses the edge metadata to specify the operand information when performing compilation of a node to an acceleration operation. For example, an accelerator might support the element-wise multiplication in ❹, but requires the number of elements being multiplied to perform the operation. Each input edge to ❸ includes the shape as part of its metadata, which allows for compilation of ❹. Domain-specific accelerators differentiate how data is stored by receiving this information from the programmer on how the variables are used. For example, the `ctrl_mdl` edge in ❶ has the `state` type modifier which causes the accelerator to store the data local to the component.



```
(a) v_temp[V] = reduce[u: u == V][v](process(e_w[u][v], v_p[u]));
    v_p[V] = apply(v_temp[V], v_p[V]);
```

Fig. 6: Graph Analytics algorithm compilation starting from (a) a PMLang program compiled to an (b) srDFG which is lowered and converted to (b) Graphicionado[14] pipeline block IR.

IV. Compilation Framework

PolyMath performs compilation in three steps: (1) compilation from PMLang to an srDFG; (2) lowering the srDFG to the granularity of the different domains and converting it to the accelerator's IR; (3) invoking the accelerator's provided compiler to generate the final binaries. Figure 6 illustrates these phases for a PMLang graph analytics implementation which is compiled to an srDFG, and then lowered and converted to Graphicionado's pipeline IR.

A. srDFG Generation

For each PMLang program, compilation forms an Abstract Syntax Tree (AST) using syntax analysis. The program AST is then traversed and a symbol table S is created, storing information contained in each component. The component information consists of variable names and variable metadata *md* (e.g., edge type, type modifier, and shape). For each component, a DFG is formed by stitching statements together using static single assignment. Lower-level, srDFG operations are formed for element-wise operations or group operations, which means edges may represent both scalar and multi-dimensional values.

After completing AST traversal generating srDFGs from PMLang statements, a single srDFG is generated starting with the highest level component `main`. Previously, component statements were skipped because all component srDFGs were not created. When the `main` *srdfg* is traversed, a component node n_c is created using previously skipped component statements and their argument type modifiers, preserving the domain annotation as $n_c.dfg.domain$ attribute. Edges adjacent to n_c are added to *srdfg* by using the type modifiers for arguments in the component signature of n_c, where type modifiers are (1) in/out edge sfor *input*/*output*, and (2) edges such that e = *(src,dst,md)* where *src* = *dst* for state. The srDFG is repeated for each component statement recursively, creating nodes and edges to generate the lower levels of operation granularity for each component, which inherit the top-level domains.

B. Example srDFG Passes

PolyMath implements a modular framework and set of APIs that enable custom, target-independent passes over the IR. These passes take an srDFG as an input and produce a transformed srDFG. This feature conveniently enables applying pipelines of passes on the same IR. Also, traditional passes such as constant propagation, constant folding, etc. are supported via this PolyMath pass infrastructure. We cover one such compiler pass below.

Algebraic combination. Transformation passes in PolyMath benefit from simultaneous access to all levels of operation granularity for a program. This bolsters traditional compiler passes such as algebraic simplification that are typically limited by single granularity IRs which hide opportunities for simplification. In contrast, a PolyMath pass can identify hidden algebraic simplifications which span multiple levels of granularity which would remain obscured in other flat IRs. As an example, if an srDFG with a top-level matrix-vector multiplication is added to the output of another matrix-vector operation contained in another node's subgraph, the matrix vector operations can be fused together by concatenating their inputs. This transformation opportunity remains unidentified in flat IRs, but PolyMath uniquely reveals these transformation prospects by preserving a program's multi-granularity in the srDFG and supporting transformations crossing granular boundaries.

C. Compilation from srDFG to Accelerator IR

Algorithm 1: srDFG Lowering Algorithm

```
function Lower (srdfg, $O_m$)
    let (N,E) = $srdfg.subDfg$
    let $O_t = O_m[srdfg.domain]$ for each $n \in N$ do
        if $n.name \notin O_t$ then
            let $subDfg$ = Lower($n, O_m$)
            $srdfg \leftarrow srdfg[n \longmapsto subDfg]$
    end
    return srdfg
```

Compiling a srDFG to a domain-specific architecture consists of (1) lowering srDFG operations supported by the target accelerator (Algorithm 1) and (2) forming valid accelerator IR by translating and combining each srDFG node (Algorithm 2). Algorithm 1 is a function Lower which takes as input a dataflow graph *srdfg* which is a pair *(N,E)* of nodes

Algorithm 2: Compilation Algorithm

```
function CompileProgram(srdfg, AccSpec)
    let $\pi_d \leftarrow \emptyset$ for $d \in Domains$
    let (N,E) = srdfg
    for each $n \in N$ do
        let $(+_d,\ m_d) = AccSpec[n.domain]$
        let $t = m_d[n.name]$
        $\pi_d = \pi_d + t(srdfg, n)$
        for each $in\_edge \in n$ do
            if ($n.domain \neq in\_edge.src.domain$) then
                $\pi_d = \pi_d + t_{load}(in\_edge, n)$
        end
        for each $out\_edge \in n$ do
            if ($n.domain \neq out\_edge.dst.domain$) then
                $\pi_d = \pi_d + t_{store}(n, out\_edge)$
        end
    end
    return $\pi_{d1}, \ldots, \pi_{dn}$
```

N and edges *E* defined in Section III-A. PolyMath lowers srDFG operations to different domains with accelerator targets that support different granularity operations. `Lower` uses a map, O_m, with domain names as keys, and lists of domain-specific accelerator operation names, O_t, as values to lower srDFG nodes to the correct granularity.

The algorithm consists of first using the *srdfg.domain* attribute as a key to determine the correct granularity of operations for lowering, storing the set of supported operations in O_t. For each node n, if *n.name* is not included in O_t, *n.srdfg* inherits the *srdfg* domain, and lowers n in *srdfg* by replacing it with a srDFG comprised of only supported operations. Replacing n in the *srdfg* consists of substituting src or dst in adjacent edges *(src,dst,md)* with a node in the *subDfg* if $src = n$ or $dst = n$. Once each $n \in N$ has been replaced with supported operations based on O_t. By preserving different levels of granularity in the srDFG, the same srDFG is capable of generating *dfg*'s with operations supported by a variety of custom accelerators. For instance, the hierarchy of RoBoX begins at the `System` level, followed by finer grained `Task` computations all the way down to varying operation granularities in it's macro dataflow graph, such as `Vector`, `Scalar`, and `Group` operations.

Once a *srdfg* has been lowered, it is compiled to an IR suitable for the accelerator using Algorithm 2. Accelerator IR for a domain d, denoted with π_d, is comprised of accelerator IR fragments, each of which is a basic operator and its arguments. To generate each π_d for the different targets, Algorithm 2 takes as input a lowered srDFG produced by Algorithm 1, and accelerator specifications for the targets corresponding to each domain. Acceleration specifications for each domain are stored in $AccSpec$, and define how srDFG nodes are translated and merged to form accelerator IR. A specification for domain d is a pair $(m_d,\ +_d)$ where:

- m_d is a map from operator names to a translation function for that operator. The translation functions, t, works as follows: given a *srdfg* and a node n, *t(srdfg,n)* returns the accelerator IR fragment π_d representing the accelerator operation for n.
- $+_d$ is an operator that combines an accelerator IR π_d and an accelerator IR fragment produced by t_d.

Having defined the necessary variables, we can describe Algorithm 2. The algorithm extracts the nodes and edges in the srDFG, then applies a translation function to each node, creating an accelerator IR fragment. Each IR fragment for each of the program's domains is separately accumulated into complete representations of an accelerator program IR, and are returned by the algorithm.

The most complicated part of the compilation are the translation functions, t. The translation function does two things: (1) identify the correct accelerator IR operation, and (2) assign the correct arguments for that operator. Assigning the correct arguments uses these steps:

1) Convert types to the equivalent accelerator type
2) Use edges with *input* type modifier as input arguments
3) Use edges with *output* type modifier as output arguments
4) Initialize IR variables for edges with the *state* type modifier
5) Add constants for arguments with *param* type modifier

If the accelerator IR fragment requires the shape of operation arguments, it is also included as part of the arguments to the operator or declaration operation. To ensure data is transferred between domain boundaries, load and store IR fragments are created when there are sources and destinations with different domains than a node. As a final step, accelerator provided compilers are used to create binaries from the generated IR.

Compilation flexibility. The combination of Algorithm 2 and 1 enables compilation to different types of domain-specific accelerator because of two key properties. First, simultaneous access to each level of srDFG recursion allows supported accelerator operations to be translated. Unsupported srDFG nodes on the particular accelerator are refined and transformed to the appropriate level of granularity through recursion which enables identification of the accelerator-supported operations. Second, the metadata stored in the srDFG allows the IR generation to be parameterized based on the target accelerator. As a result, users can create different accelerator specifications for different accelerators and these same algorithms will do the appropriate mapping. Each algorithm can be instantiated for a number of different mappings without changes to the high-level algorithm.

V. Evaluation

Table II illustrates the difference between conventional general-purpose stacks, domain-specific stacks in the literature, and PolyMath. As shown, PolyMath represent a middle ground between domain-specificity and generality, enabling cross-domain multi-acceleration.

A. Experimental Setup

1) Algorithms and Datasets.: Table III shows workloads from Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning domains and the lines of code (LOC) for the PMLang implementation. Table IV shows a break down of end-to-end application domains, algorithms, configurations, and PMLang LOC.

Single domain workloads. In *Robotics* domain, we have two benchmarks, MobileRobot [21] and Hexacopter [22]. Section II discusses the two-wheeled MobileRobot in detail. Hexacopter is a six-rotor micro UAV that uses motion planning and orientation control to determine trajectory. For both these workloads the physical robot and task specification is expressed in PMLang. For *Data Analytics* we have Low Rank Matrix Factorization (LRMF) and Kmeans clustering. LRMF converts a large matrix into two smaller matrices, which if taken product of, represent the original matrix. For LRMF we use two Movielens [23] datasets. Kmeans clustering partitions data into k-clusters. For one Kmeans workload cluster hand written digits with mnist [24] dataset. The second benchmark uses data from the UCI repository [25] to cluster households with similar electricity consumption. In the *Digital Signal Processing* domain we have four benchmarks, two for each Fast Fourier Transform (FFT), and Discrete Cosine Transform (DCT). The FFT implementation is a fine-grained butterfly and bit-reversal to transform a signal to frequency domain. The DCT algorithm applies a filter kernel to an input image and is used for compression. For *Deep Learning*, we use two popular convolutional neural networks, ResNet-18 [26] and MobileNet [27] for object classification. For *Graph Analytics*, we implement and apply Breadth-First Search on two graphs, one of Twitter users and followers [28] and another of Wikipedia links [29].

End-to-end cross-domain applications. Table IV shows the end-to-end applications for our case study, the different domains they comprise of, and specification of each algorithm. The brain stimulation application,`BrainStimul`, is described in Section II. The stock market application, called `OptionPricing`, predicts call option price in stock market and uses two data analytics algorithms. This application first performs sentiment analysis through logistic regression on news articles to understand market signals and then Black-Scholes to predict the price.

2) Optimized CPU and GPU implementations.: Table V and VI shows the optimized CPU and GPU framework their specifications. The robotics's CPU implementation uses ACADO Toolkit [30] to implement optimized, self-contained C code and uses cuBLAS [31] libraries for GPUs. For graph analytics, we used Intel GraphMat [32] for CPU implementations and Enterprise [33] for GPU. The DSP workloads use

TABLE II: A comparison of computational stacks.

Domain	General-Purpose Processors	Graphicionado [14]	Darwin [18]	DNNWeaver [19]	TVM [16]	TABLA [17]	RoboX [15]	DeCO [7]	BCP Acc [20]	PolyMath
Robotics	✓	✗	✗	✗	✗	✗	✓	✗	✗	✓
Graph Analytics	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓
DSP	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
Data Analytics	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
Deep Learning	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓
Genomics	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SAT Solvers	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗

TABLE III: Benchmarks and workloads used to evaluate PolyMath.

Domain	Benchmark	Algorithm	Config/Dataset	PMLang LOC
Robotics	**Mobile Robot**	Model Predictive Control	Trajectory Tracking, Horizon = 1024	52
	Hexacopter	Model Predictive Control	Altitude Control, Horizon = 1024	197
Graph Analytics	**Twitter Followers**	Breadth-First Search	#Vertices=61.57M, #Edges=1468.36M	14
	Wikipedia Links	Breadth-First Search	#Vertices=3.56M, #Edges=84.75M	14
	LiveJournal	Single Source Shortest Path	#Vertices=4.84M, #Edges=68.99M	14
Data Analytics	**MovieL (100k)**	Low Rank Matrix Factorization	1682 movies, 943 users; 100000 ratings	43
	MovieL (20M)	Low Rank Matrix Factorization	40110 movies, 259137 users; 244096 ratings	43
	DigitCluster	K-Means Clustering	784 features;120000 images;K=10	41
	ElecUse	K-Means Clustering	4 features; 2075259 data points; K=12	41
DSP	**FFT-8192**	Fast-Fourier Transform	1D FFT-real; 8192x1 input	12
	FFT-16384	Fast-Fourier Transform	1D FFT-real; 16834x1 input	12
	DCT-1024	Discrete Cosine Transform	1024x1024 image; 8x8 kernel, stride=8	31
	DCT-2048	Discrete Cosine Transform	2048x2048 image; 8x8 kernel, stride=8	31
Deep Learning	**ResNet-18**	Deep Neural Network	Batch Size = 1, ImageNet	117
	MobileNet	Deep Neural Network	Batch Size = 1, ImageNet	102

TABLE IV: Algorithmic composition of end-to-end applications.

Benchmark	Algorithm	Domain	Config/Dataset	LOC
Brain Stimul	Fast-Fourier Transform (FFT)	DSP	1D FFT, 4096 Input	12
	Logistic Regression (LR)	Data Analytics	4096 features	8
	Model Predictive Control (MPC)	Robotics	Horizon = 1024	64
Option Pricing	Black-Scholes (BLKS)	Data Analytics	8192 options	10
	Logistic Regression (LR)	Data Analytics	129549 words	8

TABLE V: Domains and accelerators used for evaluations.

Domain	PolyMath Accelerator	Baseline Framework
Robotics	RoBoX (ASIC)	ACADO/cuBLAS
Graph Analytics	GRAPHICIONADO (ASIC)	Intel GraphMat/Enterprise
Data Analytics	HyperStreams(FPGA) / TABLA (FPGA)	MLPack/OpenBlas/CUDA
DSP	DECO (FPGA)	FFTW3/cuFFT/NVIDIA-DCT
Deep Learning	TVM-VTA (FPGA)	TVM/Tensorflow

TABLE VI: CPU, FPGA, and ASIC specifications.

	CPU	FPGA	ASIC	GPU
Chip	Xeon E-2176G	UltraScale KCU1500	RoBoX/ GRAPHICIONADO	Titan Xp/ Jetson AGX Xavier
Cores	6	-	-	3,840/ 512
Memory/ BRAM	128GB	75MB	512KB/ 64MB	12 GB/32 GB
Power	80W	35W	3.4W/7W	250W/30W
Frequency	3.7 GHz	150MHz	1GHz/ 1GHz	1.5GHz/ 1.3GHz
Logic Tables	-	1,451	-	-
Compute Units	-	5,520	256/8	-

C subroutine libraries [34, 35] for CPU and Nvidia implementations [36, 37] for GPU. For data analytics, we use mlpack [38], a fast and flexible C++ ML library built on top of OpenBLAS [39], NVBLAS [40], and Armadillo [41]. The Deep Learning workloads are compiled using optimized Tensorflow [42] for CPU and GPU. All of our experiments were performed on an Intel Xeon E7, Titan Xp GPU, and low-power Jetson Xavier AGX.

3) Domain-Specific accelerators.: Table V shows the accelerator used for each domain. To evaluate each benchmark on domain-specific accelerators, PolyMath was used to compile programs to the target accelerator IR, and the target accelerator's compiler was used to generate executable binaries. RoBoX's [15] Macro DFG from srDFG as it offers programmable ASIC for system, tasks, and penalties for control algorithms optimized using MPC. We use GRAPHICIONADO [14], an ASIC accelerator for graph analytics algorithms expressed as vertex programs, as the target for the Graph Analytics workloads. We compile FFT and DCT srDFG representation to DECO [7], a DSP block based FPGA accelerator, by translating to it's DFG, which is then compiled as executable binaries. For LRMF and kmeans we convert srDFG to TABLA [43], an open source template-based FPGA accelerator for machine learning, by compiling a srDFG to a DFG.We use TVM-VTA [44], a programmable deep learning FPGA accelerator, as the target for Deep Learning workloads, as it is state-of-the-art and open-source. Each DSA requires specific levels of operation granularity: single operation [7, 17], coarse DNN layers [44], and coarse time snapshots [15], enabled by each srDFG's multi granularity, for mapping of kernels to the accelerator.

Multi-acceleration. For `BrainStimul`, we compile parts to DECO [7] (FFT-4096), TABLA [17] (Logistic Regression), and RoBoX [15] accelerators. For `OptionPricing`, we execute logistic regression based sentiment analysis on TABLA [17] and Black Scholes on HyperStreams [45]. All accelerators are cascaded as a single System On Chip (SOC), comprised of memory and a host. A light-weight manager executes on the host, ensuring data dependencies between different accelerators and initiating DMA transfers between DRAM and local accelerator memory. This setting is similar to prior work [46] that also uses an array of micro-accelerators.

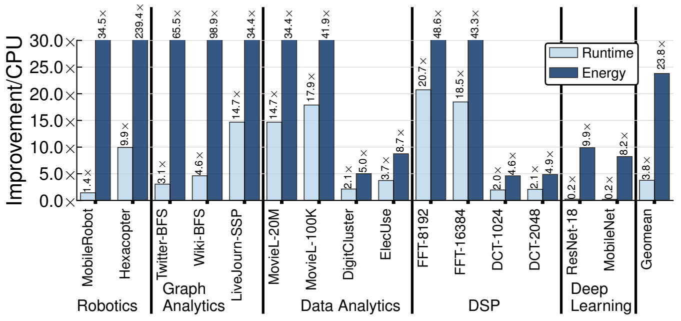


Fig. 7: Runtime and Energy improvement of PolyMath over CPU.

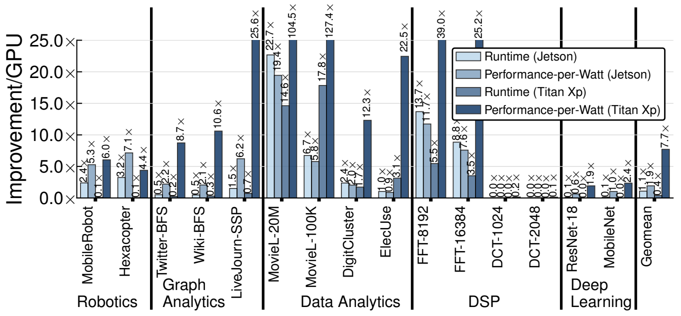


Fig. 8: Runtime and Performance-per-Watt improvement of PolyMath over GPU.

B. Experimental Results

The goal of PolyMath is to facilitate the use of the wide variety of custom accelerators across end-to-end cross domain applications. It does so by abstracting away hardware level details through a versatile, extensible, and a modular stack that can maintains the required levels of kernel granularities best suited for each design. In this section we compare domain-specific accelerators executing PMLang code with optimized CPU and GPU implementations to better understand the portability of PolyMath. We then perform case studies through two end-to-end applications and observe that PolyMath allows cross domain multi-acceleration.

1) Performance and Energy Comparisons: ***Single kernel comparisons.*** Figure 7 and Figure 8 show the speedup of PolyMath compiled programs listed in Table III to domain specific accelerators over Xeon E-2176G CPU, Titan Xp GPU, and Jetson Xavier AGX as the baseline, respectively. On average, PolyMath translated implementations outperform Xeon E-2176G and Jetson Xavier by 3.3× and 1.2× in terms of runtime and offer 7.2× and 1.7× more Performance-per-Watt over Titan Xp and Jetson Xavier GPUs. Smaller benchmarks such as MovieLens-100K and ElecUse are unable to fully utilize Titan XP, thus cannot obtain higher benefits in comparison to Jetson but incur higher PPW. The average still demonstrates an increase in both performance and energy in the cross-domain setting. PolyMath implementations also offer 18.1× more energy efficiency over CPUs, but due to many lower power accelerator backends only offers 40% of the GPU performance. This is especially true for discrete cosine transform and deep learning benchmarks; DCT due to its high coarse granular matrix multiplications for which DECO a programmable FPGA accelerator is not as effective as Titan Xp and deep learning models because our backend for CNNs is VTA that is designed as a low-power accelerator but is also being compared to a high-end GPU for uniformity. Note that PolyMath does not contribute any overhead specifically for deep learning acceleration because it offers direct conversion of srDFG to the TVM nodes.

Optimal performance comparison. The accelerators used for execution of PolyMath implementations also offer custom stacks that are built for their target architecture. We compare PolyMath to implementations in their native stack, which represent optimal executions to demonstrate the cross-domain overhead of our stack to native implementations. Figure 9 compares the performance of PolyMath implementations with the optimal performance that can be reached by programs written by experts for each of the accelerators. The figure shows that PolyMath achieves 83.9% the optimal runtime.

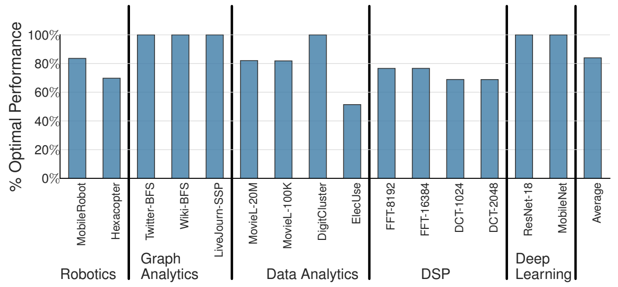
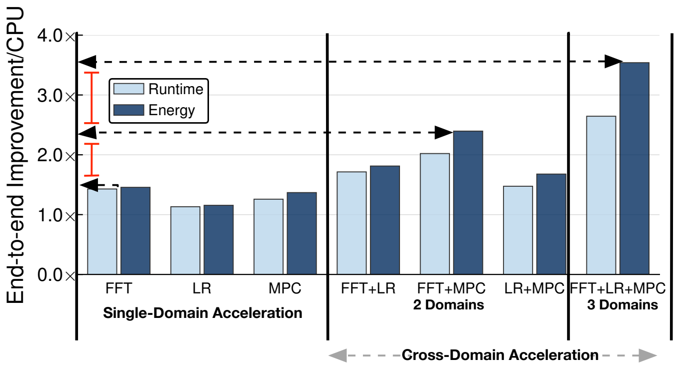
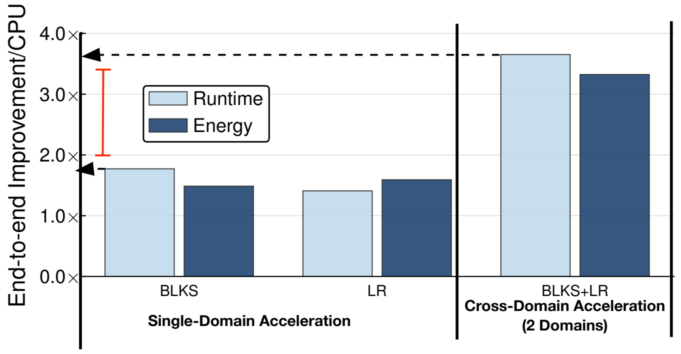


Fig. 9: Percent of optimal runtime for PolyMath translated implementations compared to hand-tuned implementations.

The performance of PolyMath relative to optimal implementations is dependent on the domain and the algorithm because each stack differs in the level of data semantics that can be complex to compile to from a more expressive language. For instance, accelerators specializing in Deep Learning often utilize three primary type modifiers for variables in neural network graphs: input, output, and weights–each of which can be directly mapped to type modifiers in PMLang, thus incur zero overhead. In contrast, Robotics algorithms contain unique data semantics, such as task penalties, constraints variables, time varying references, etc., which do not differentiated with PolyMath type modifiers, thus implementations do not reach optimal performance. Accelerators such as DECO require specific topologies for their graph-based IR, i.e. balanced DFGs, because they rely on stage-based computation, which results in reduced execution time relative to PolyMath translations. In the case of Data Analytics, we see a low percentage of optimal performance for ElecUse, because the benchmark is small, which makes any extra operations included in the srDFG have a more significant impact on performance relative to the optimal implementation. In contrast, DigitCluster uses the same algorithm but on a larger dataset, thus can amortize the overhead more effectively. It is important to note that the optimal performance is reached by a specialized program on each accelerator written by an expert, whereas PolyMath offers support for multiple domains constituting end-to-end applications that can be expressed as single comprehensive program, coupled with *sr*DFG to pave way for compilation to multiple accelerators.



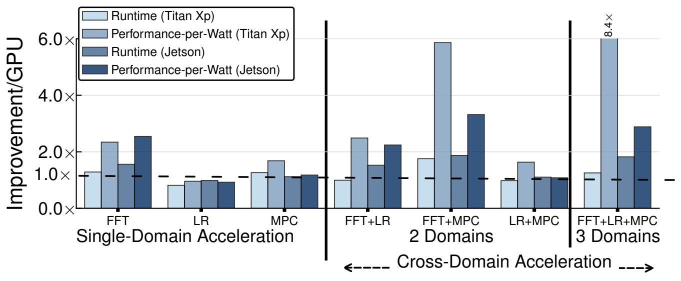
(a) `BrainStimul` brain stimulation application



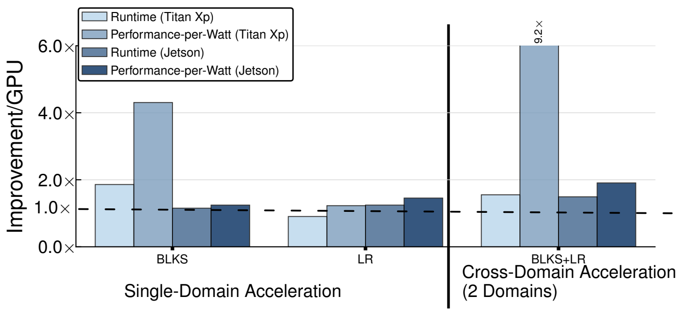
(b) `OptionPricing` finance application

Fig. 10: Runtime and energy improvement over CPU of end-to-end applications for different combinations of accelerated domains.

2) End-to-End Application Case Study: PolyMath offers means to express cross domain applications as a single program which can be compiled to multiple accelerators pertaining to each of these domains. Figure 10 shows the runtime and energy improvement of the end to end applications in comparison to CPU. Figure 11 illustrates the runtime and performance-per-Watt improvement in comparison to Titan Xp and Jetson. Figure 10a and Figure 11a shows these results for `BrainStimul` and Figure 10b and Figure 11b for `OptionPricing` application. In these graphs we provide entire application improvement for all possible acceleration combinations, from one domain algorithm accelerated to cross-domain where all algorithms are accelerated. Each end-to-end result incorporates data communication overheads from data transfer between hardware. Stand-alone kernel acceleration, as shown in , can offer very high speedups. However, when these kernels are incorporated within a more comprehensive application, those speedups do not manifest in the entire application because the non-accelerated kernel becomes a bottleneck. For instance, the gap between the highest benefit obtained from the best single-domain acceleration and cross-domain end-to-end acceleration, is 1.85× for `BrainStimul` and 2.06× for `OptionPricing` (Figure 10). Every kernel that is added for acceleration not only benefits itself from specialized execution but also reduces the Amdahl's burden and magnifies other accelerated component's impact. The benefits of individual kernel acceleration are present despite end-to-end communication runtime overheads of 23.4% and 17.0% and energy overheads of 21.8% and 12.4% for `BrainStimul` and `OptionPricing`, respectively.



(a) `BrainStimul` brain stimulation application



(b) `OptionPricing` finance application

Fig. 11: Runtime and Performance-per-Watt improvement over GPU for combinations of accelerated domains for end-to-end applications.

Figure 11a and 11b show the `BrainStimul` and `OptionPricing` results for both Titan Xp and Jetson. Figure 11a shows that PolyMath offers 1.2× runtime improvement over Titan Xp compared to 1.8× over Jetson. In contrast, PolyMath improves performance-per-watt by 8.3× over Titan Xp compared to 2.8× for Jetson due to its lower power consumption. As Figure 11b shows, the `OptionPricing` benchmark underutilizes the Titan Xp, only offering 1.5× and 9.2× improvement in performance and performance-per-watt compared to 1.4× and 1.9× over Jetson. This is caused by the difference in levels of coarse parallelism in the algorithms. Overall, the PolyMath implementation of `OptionPricing` still outperforms both GPUs for both runtime and performance-per-watt.

Lastly, Figure 12 shows end-to-end Polymath implementations achieve 76.7% optimal performance for `BrainStim` and 76.9% for `OptionPricing` compared to entirely manual implementations of each application. Given the fact that PolyMath offers greater ease of programming compared to Python (Figure 13), the automation overhead of 23.1% is a fair tradeoff.

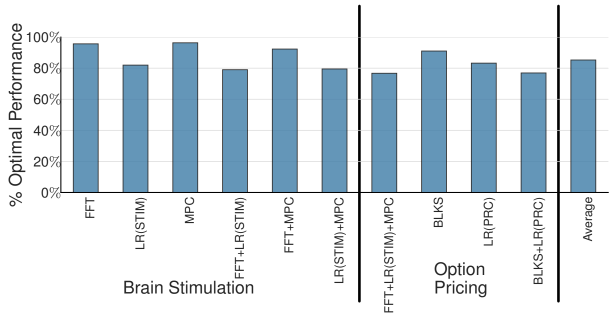
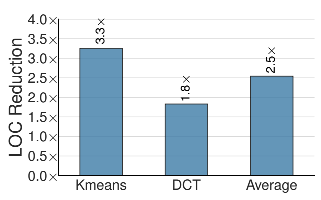
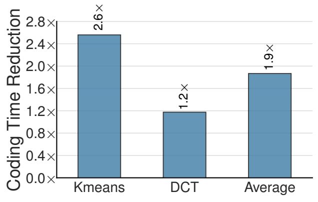


Fig. 12: Percent of optimal performance for `BrainStim` and `OptionPricing` compared to hand-tuned implementations.



(a) Lines of Code Reduction



(b) Coding Time Reduction

Fig. 13: Reduction in Lines of Code (LOC) and coding time with PMLang over Python for Kmeans and DCT.

3) User Study: To determine the usability of PMLang, we conducted a user study with 20 programmers who are either professional software engineers or PhD students in computer science. The goal of the user study is to measure the expressiveness of PMLang by comparing it to Python, an intuitive programming language which is commonly used in three of the focus domains of PolyMath: DSP, Data Analytics, and Deep Learning. The study tasked each user in the study with implementing either a DSP or Analytics algorithm in Python or PMLang. The users were divided into four groups; Kmeans in Python, Kmeans in PMLang, DCT in Python, and DCT in PMLang. For fairness and ease in expression of tensor algebraic operations, we allow the users to import Python modules such numpy [47]. Each participant in the user study has varying levels of expertise (from beginner to proficient) in the target domains, and is proficient in Python. Every participant went through the following process:

1) Participants were introduced to PMLang with a short, six-minute video which walked through the language and small examples.
2) To avoid any algorithm knowledge bias, participants were randomly assigned either the DSP or ML algorithm to implement in either Python and PMLang.
3) To minimize variation in algorithm understanding, users were not allowed to begin their first implementation before having read and confirmed their understanding of the algorithm.
4) We timed participants during their implementations and measured their Lines of Code (LOC) after completion.

Results. Figure 13 compares the LOC between Python and PMLang as a ratio of Python LOC to PMLang LOC in (a), and the implementation time of Python and PMLang as a ratio of Python implementation time to PMLang implementation time in (b). The results show that PMLang required 2.5× fewer lines of code on average and 1.9× less implementation time on average. The Kmeans implementation averaged 3.3× fewer LOC, whereas for DCT the average reduction of LOC is 1.8×. In general, the Kmeans algorithm is more verbose than DCT and on average required 47.6% more lines of Python code than DCT. Because there were more lines of code included in Kmeans, there was more opportunity to reduce multi-line operations to a single PMLang statement, which explains the difference in LOC reduction.

The greater complexity of Kmeans appears to have an effect in the speedup of implementation time as well, where the average speedup for Kmeans was 2.6× and the average speedup for DCT was 1.2×. Part of this speedup can be attributed to typing more LOC in PMLang, but it is also a result of being able to directly translate mathematical notation to the equivalent PMLang statement. These results indicate that the more complicated the mathematical program is, the more the programmer will benefit from implementing the program in PMLang. Further, PMLang is expressive enough for programmers unfamiliar with the language to write algebraic expressions more efficiently than they would write the same expressions in a language they are familiar with, Python.

VI. Related Work

DSLs for custom architectures. There are various domain-specific languages designed to facilitate the use of hardware accelerators. These languages are mostly designed for a single domain [48–51] or like Spatial [52], they focus on conveniently expressing lower level hardware-centric information. Another language, Halide [50], allows expression of image processing pipelines and contains constructs for filter-based algorithms. Lime [53] focuses on high-level synthesis from Java, thereby enabling execution of Java programs on both FPGAs and CPUs. Instead, PolyMath offers a *Cross-Domain Language (CDL)* and compute stack to explore the emerging tradeoff between expressiveness and performance while leveraging currently isolated, domain-specific accelerators.

Mathematical and scientific computing environments. There are numerous scientific and numerical programming environments [54–57], and frameworks [47, 58]. PolyMath uniquely provides a 1-to-1 mapping of mathematical expressions to its statements and leverages the natural parallelism in formulas without any explicit annotations for vectorization In contrast, MATLAB [54], Julia [55], or R, require manual effort from the user to identify the parallelism across different computations, vectorize its code, and determine column/row arrangements for matrix operations. Moreover, these languages do not delineate between the semantics of data in their programs and do not offer a multi-granular representation, as offered by PolyMath, to enable usage of various domain-specific accelerators.

Intermediate Representations. A number of intermediate representations [59, 60] provide abstractions to enable program analysis using virtual resources. Both LLVM and JVM operate at the granularity of a single CPU instruction, which is highly inefficient for domain-specific architectures. Some works [61] have adapted LLVM to guarantee independence between parallel operation threads by using a dataflow graph structure intended for heterogeneous platforms. A number of other works [16, 62–66] focus on the domain of machine learning and have implemented an end-to-end approach for optimization on heterogeneous platforms after performing optimizations from a high-level language. These works supported limited algorithm domains [16, 62–65], and rely on C/C++ or other general-purpose programming languages [61, 66], requiring the programmer to express complex mathematical expressions in unintuitive ways. MLIR [66] is a hierarchical, high level IR, but is general-purpose and as such is on the end of the expressiveness curve (Figure 1). Whereas PolyMath is restricted by its mathematical language (PMLang) to a limited set of domains, falling in the middle of the spectrum of expressiveness. Furthermore, MLIR does not have any compilation stack to support variety of accelerators from different domains as PolyMath does and is practically demonstrated in the evaluations. Tiramisu [67] introduces a scheduling language with novel commands to explicitly manage the complexities that arise when targeting multicores, GPUs, and distributed machines. Tiramisu offers an IR based on the polyhedral model to allow fine-grained optimization. As such, Tiramisu can serve as a potential backend for PolyMath that deals with the higher-level complexity of expressing cross-domain application and not low-level fine-grained optimization.

Acceleration frameworks and toolchains. TensorFlow [42] is an end-to-end open source platform for expressing ML algorithms in Python. Deep learning accelerators (e.g., TPU [13]) leverage Tensorflow. Similarly, a variety of deep learning frameworks [19, 68, 69] allow users to run their DNNs on FPGA based hardware designs. Full stack solutions such as TABLA [17] and RoBoX [15] support classical supervised machine learning and model predictive control in robotics, respectively. Other toolchains [70–73] aim to simplify running deep neural networks on hardware accelerators by performing design space exploration to find the best configuration for their particular design. These solutions, however, are bound to their own custom architectures for particular platforms (FPGAs or ASICs). In contrast, the *sr*DFG offers a flexible hook that can be translated to these toolchains and frameworks as well as to future accelerator designs and platforms. The cross-domain nature of PolyMath that supports Robotics, Graph Analytics, DSP, Data Analytics and Deep learning sets it apart from these domain-specific stacks.

VII. Conclusion

As domain-specific accelerators are becoming prevalent, there is an emerging tradeoff between expressiveness and performance. This paradigm–a pendulum swing from general-purpose processing to the opposite direction–creates implicit programming silos between different domain. This paper set out to explore the region between these extremes and explore the new expressiveness-performance tradeoff. To that end, we defined a *cross-domain* computational stack, PolyMath, that bridges the expressiveness gap between multiple domains, Robotics, Graph Analytics, DSP, Deep Learning, and Data Analytics. The results from user study and performance evaluations showed that PolyMath strikes an effective balance between expressiveness and performance while enabling cross-domain multi-acceleration. It is time to look beyond the timely, yet temporary, success of domain-specific accelerators and devise a future that enables end-to-end applications. The current approach towards acceleration excludes significant opportunities by restricting the domain. To harness these untapped opportunities, a new paradigm needs to emerge that breaks the boundaries of domains, but also preserves the benefits of domain-specificity. PolyMath takes the initial step in breaking this new ground.

VIII. Acknowledgments

This work was in part supported by generous gifts from Qualcomm, Google, Microsoft, Xilinx, Leidos as well as the National Science Foundation (NSF) awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, National Institute of Health (NIH) award #R01EB028350, and AirForce Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA) under agreement number #FA8650-20-2-7009 and #HR0011-18-C-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding 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 Qualcomm, Google, Microsoft, Xilinx, NSF, AFSOR, NIH, AFRL, DARPA or the U.S. Government.

References

[1] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016. doi: 10.1109/MICRO.2016.7783748.

[2] Texas instruments c6000tm dsp, 2007. URL http://www.ti.com/processors/digital-signal-processors/c6000-floating-point-dsp/overview.html.

[3] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[4] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[5] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[6] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[7] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. Deco: A dsp block based fpga accelerator overlay with low overhead interconnect. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, May 2016. doi: 10.1109/FCCM.2016.10.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[9] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. Encodeep: Realizing bit-flexible encoding for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(6):1–29, 2020.

[10] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic. In *PACT*, 2020.

[11] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[12] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO*, October 2020.

[13] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. URL http://arxiv.org/abs/1704.04760.

[14] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016. doi: 10.1109/MICRO.2016.7783759.

[15] Jacob Sacks, Divya Mahajan, Richard C. Lawson, and Hadi Esmaeilzadeh. Robox: An end-to-end solution to accelerate autonomous control in robotics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 479–490, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 978-1-5386-5984-7. doi: 10.1109/ISCA.2018.00047. URL https://doi.org/10.1109/ISCA.2018.00047.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/chen.

[17] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. March 2016.

[18] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 199–213, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173193. URL http://doi.acm.org/10.1145/3173162.3173193.

[19] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. October 2016.

[20] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *Proceedings of the 45th Annual Design Automation Conference*, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581156. doi: 10.1145/1391469.1391669. URL https://doi.org/10.1145/1391469.1391669.

[21] Felipe Kuhne, Joao Manoel Gomes da Silva Jr, and Walter Fetter Lages. Mobile robot trajectory tracking using model predictive control. In *Latin American Robotics Symposium*, 2005.

[22] M. Achtelik M. Kamel, K. Alexis and R. Siegwart. Fast nonlinear model predictive control for multicopter attitude tracking on so(3). In *IEEE Multi-Conference on Systems and Control*, 2015.

[23] Grouplens. Movielens dataset. URL http://grouplens.org/datasets/movielens/.

[24] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[25] M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605587998. doi: 10.1145/1772690.1772751. URL https://doi.org/10.1145/1772690.1772751.

[29] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[30] Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. Acado toolkit—an open-source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011. doi: 10.1002/oca.939. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/oca.939.

[31] Nvidia. Dense linear algebra on gpus, . URL https://developer.nvidia.com/cublas.

[32] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015. ISSN 2150-8097. doi: 10.14778/2809974.2809983. URL https://doi.org/10.14778/2809974.2809983.

[33] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[34] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93 (2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[35] Nvidia. Nvidia toolkit, . URL https://developer.nvidia.com/cuda-toolkit.

[36] Nvidia. Nvidia cuda fast fourier transform library, . URL https://developer.nvidia.com/cufft.

[37] Nvidia. Nvidia cuda sdk - image/video processing and data compression, 2008. URL https://www.nvidia.com/content/cudazone/cuda_sdk/Image_Video_Processing_and_Data_Compression.html.

[38] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[39] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 684–691, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4903-3. doi: 10.1109/ICPADS.2012.97. URL http://dx.doi.org/10.1109/ICPADS.2012.97.

[40] Nvidia. Nvidia nvblas library., . URL http://docs.nvidia.com/cuda/nvblas.

[41] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. In *Journal of Open Source Software*, 2016.

[42] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015. URL http://download.tensorflow.org/paper/whitepaper2015.pdf.

[43] ACTLab. TABLA source code, 2017. http://www.act-lab.org/artifacts/tabla/.

[44] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Lianmin Zheng, Eddie Yan, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, July 2019.

[45] G. W. Morris and M. Aubury. Design space exploration of the european option benchmark using hyperstreams. In *2007 International Conference on Field Programmable Logic and Applications*, pages 5–10, 2007.

[46] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. A. Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini. Exploring architectural heterogeneity in intelligent vision systems. In *2015 IEEE*

21st International Symposium on High Performance Computer Architecture (HPCA), pages 1–12, 2015.

[47] Stéfan Van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011. URL http://arxiv.org/abs/1102.1523.

[48] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems*, 2015.

[49] Mirko Bordignon, Kasper Stoy, and Ulrik Pagh Schultz. Generalized programming of modular robots through kinematic configurations. In *International Conference on Intelligent Robots and Systems*, 2011.

[50] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017. ISSN 0001-0782. doi: 10.1145/3150211. URL http://doi.acm.org/10.1145/3150211.

[51] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 609–616, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5. URL http://dl.acm.org/citation.cfm?id=3104482.3104559.

[52] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 296–311, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192379. URL http://doi.acm.org/10.1145/3192366.3192379.

[53] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869469. URL http://doi.acm.org/10.1145/1869459.1869469.

[54] *MATLAB version 9.3.0.713579 (R2017b)*. The Mathworks, Inc., Natick, Massachusetts, 2017.

[55] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL http://arxiv.org/abs/1209.5145.

[56] Guy L. Steele, Jr. Parallel programming and code selection in fortress. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122972. URL http://doi.acm.org/10.1145/1122971.1122972.

[57] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. URL https://www.R-project.org/.

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[59] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[60] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390590X, 9780133905908.

[61] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 68–80, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4982-6. doi: 10.1145/3178487.3178493. URL http://doi.acm.org/10.1145/3178487.3178493.

[62] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: an open hardware-software stack for deep learning. *CoRR*, abs/1807.04188, 2018. URL http://arxiv.org/abs/1807.04188.

[63] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 58–68, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5834-7. doi: 10.1145/3211346.3211348. URL http://doi.acm.org/10.1145/3211346.3211348.

[64] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. URL http://arxiv.org/abs/1802.04730.

[65] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for

neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907.

[66] Chris Lattner and Jacques Pienaar. Mlir primer: A compiler infrastructure for the end of moore's law, 2019.

[67] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018. doi: 10.1109/ASAP.2018.8445108.

[68] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, , Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38:8–20, March 2018. URL https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/.

[69] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. ACM, June 2018. URL https://www.microsoft.com/en-us/research/publication/a-configurable-cloud-scale-dnn-processor-for-real-time-ai/.

[70] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016. doi: 10.1109/ISCA.2016.32.

[71] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.

[72] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, 2020.

[73] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. pages 97–108, 06 2014. ISBN 978-1-4799-4394-4. doi: 10.1109/ISCA.2014.6853196.