PolyMath Documentation

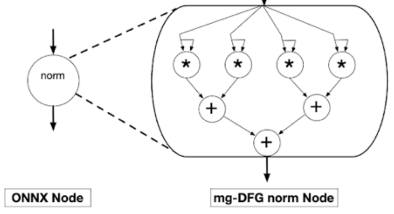
PolyMath Overview

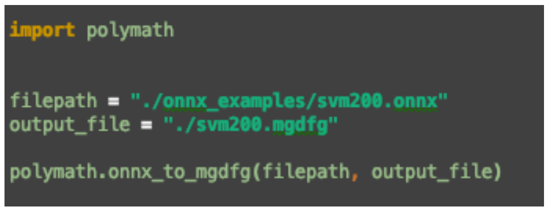
Starting with an ONNX description of an ML algorithm, we create an intermediate representation (IR) in the form of a multi-granular dataflow graph (*mg*-DFG) that is more conducive to translation to a Verilog implementation. PolyMath is the framework which relies on the mg-DFG for compilation.

We adopt the widely supported Open Neural Network Exchange (ONNX) format in order to maximize interoperability across various programing environments. To encapsulate varying granularities of operations, we devise a multi-granular Dataflow Graph (*mg*-DFG) which is recursively defined with the *mg*-DFG nodes.

The multi-granular representation facilitates optimization in several ways, e.g., (1) utilizing optimizations that are pre-developed for certain complex operations (e.g., building a binary tree for the L2 norm or optimizing the flow of data for convolution); (2) simultaneously, preserving the capability to perform fine-grained scheduling and mapping optimization; and (3) enabling support for future innovative operators that ML experts (e.g., from the NSF RTML program) may innovate as they will be represented as new *mg*-DFG subgraphs in the IR that can be put through scheduling and mapping compiler optimization passes.

To translate an ONNX file into a multi-granular Dataflow Graph (*mg*-DFG), we utilize the ONNX python framework to traverse the ONNX graph which is comprised of nodes representing coarse-grained ML operations on multi-dimensional arrays of input data. During traversal, *mg*-DFG nodes and edges are generated using the attributes of each ONNX operation and its inputs/outputs. The operations which comprise each coarse-grained operation (e.g., multiply-adds form a norm operation, as shown in the figure below) are added to each *mg*-DFG node using instantiations of pre-defined templates. Figure 1 demonstrates the translation of a "norm" ONNX node to the equivalent *mg*-DFG node.




```
import polymath

filepath = "./onnx_examples/svm200.onnx"
output_file = "./svm200.mgdfg"

polymath.onnx_to_mgdfg(filepath, output_file)
```

Figure 1: Translation of a "norm" ONNX node to the equivalent *mg*-DFG node which contains all the fine-grained operations that form a "norm" operation..

Flexible Compilation Targets Using PolyMath

PolyMath's compilation flexibility enables support for different types of accelerator. Increasing flexibility was enabled by the completion of two additional features in the VeriGOOD-ML compilation pipeline: (1) the implementation of a reusable hardware abstraction called a hierarchical architecture graph (HAG) and corresponding architecture description language embedded in Python for targeting different types of accelerators with a unified interface and (2) a series of compilation passes which use a HAG for mapping, scheduling, and optimizing programs for different target accelerators. The overall scheme is illustrated in Figure 2. Because we have already implemented a reusable, multi-granular DFG (*mg*-DFG) as an IR for both DNN and non-DNN algorithms along with compiler pass infrastructure, the remaining task for end-to-end compilation is introduction of hardware-specific attributes to the compilation pipeline. Further, by consolidating different types of hardware specific attributes to a single abstraction, the end-to-end compilation pipeline can be reused for different types of accelerator as shown in the figure below. The only requirement for compilation to new hardware targets is that hardware designers devise a HAG, which can be done through the architecture description language we have created.

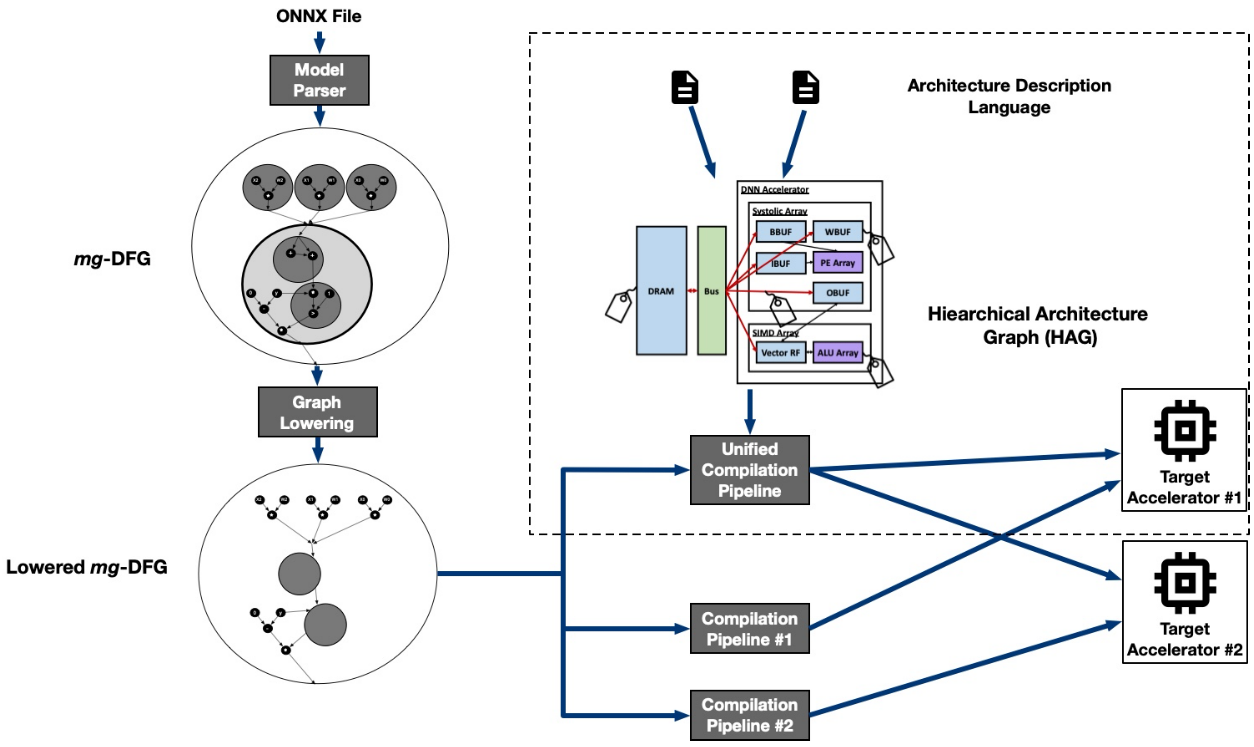


Figure 2: The concept of using an architecture description language (ADL), modeled by a hardware architecture graph (HAG) to map the *mg*-DFG to hardware.

Our customizable compiler backend enables scheduling, mapping, coordination of operations in an *mg*-DFG and their communication with on-chip and off-chip resources which flexibly allows for a single, customizable, compilation pipeline for different types of architectures (TABLA, GeneSys, Axiline) along with different types of optimization passes. Our central idea is the novel concept of Hierarchical Architecture Graphs (HAGs) which are created using an Architecture Description Language embedded in Python. Each HAG is comprised of three types of nodes: HAG node for computations, HAG nodes on and off-chip communication, and HAG nodes for storage.

The compilation process consists of traversing an *mg*-DFG, and iteratively mapping, scheduling, and compiling the nodes to the HAG. During this process, the attributes of HAG nodes can be used to optimally select compilation parameters such as tiling, node fusion, unrolling, etc. That is to say, the pass will consider previous and next layers to decide how to weave the codelets together with the target HAG. This pass will be achieved by a parameterized graph optimization algorithm that takes as input a series of node operations for fusing together, with each set of operations being broken into slices of the DFG that can be mapped and scheduled on the HAG. By dividing the compilation, the scope of optimization to find the best mapping becomes tractable but then the slices need to be stitched together. Each of these fused sets of DFG nodes will be stitched together using another compilation pass which will utilize the off-chip communication nodes and on-chip communication nodes within the HAG to include data transfer and communication between fused sets of DFG nodes.

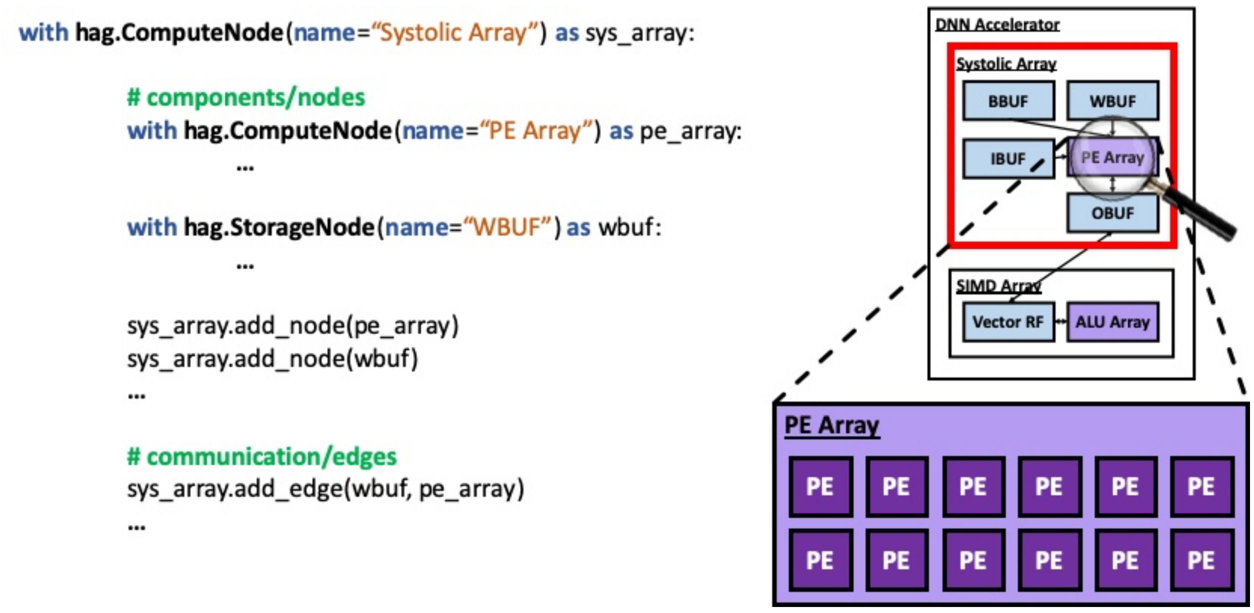
Architecture Description Language and Hierarchical Architecture Graph Architecture description and compiler optimization has an intimate relationship which makes it natural to start off by building an abstraction for describing various architectures in the system. By combining both insights from our previous work on balancing generalization and specialization by leveraging algorithmic commonalities from various domains in the *mg*-DFG, we have developed an Architecture Description Language (ADL) in the form of a Python-embedded language that can depict the essence of each architecture, and, possibly, the variations of microarchitecture across multiple domains. By building such abstraction, it enables the compiler to expand its capability from optimizing for single piece of hardware to a heterogeneous computing environment where there are multiple disparate processors and accelerators. This ADL is built on top of Python to improve usability and versatility, easily working in tandem with various machine learning frameworks. To represent diverse types of accelerator, there are four primary attributes which must be included in the abstraction.

- **Hierarchy Representation**: Accelerators are composed of multiple levels of hierarchy which can be as fine grained as a single ALU, and as coarse grained as an entire systolic array consisting of buffers, an array of processing engines (PEs), and interconnect. Using multiple levels of hierarchy makes architectural composition more tractable, extensible and simplifies compilation by relieving the user of compiling coarse operations to fine grained ALU operations while preserving access to such fine-grained architecture attributes.
- **Specification of compute, storage, and communication components**: It is vital to understand that without providing abstractions for on-chip and off-chip data transfer, any generalized compilation effort for domain-specific architectures is an exercise in futility. In fact, the benefits and power of these specialized hardware comes from tightly coordinating fine-grained calculation with low-overhead operand delivery. Therefore, we highlight not only compute nodes such as processing engines, but also the means of communication and data transfer through storage and compute nodes as part of the HAG abstraction. Within the different categories of architecture component, there are a number of attributes which must be accounted for to successfully compile DFG nodes. As an example, storage nodes have a pre-defined capacity for the amount of data they are able to

store, which needs to be accounted for when determining the size of tiling for a given operation.

- **Listing capabilities**: Each component within a HAG supports different types of capabilities, which are used to identify which DFG nodes can be mapped to which HAG components. In addition, capabilities help identify how to compile a certain operation using the supported capabilities of an architecture's nodes. Each architecture description is compiled to a hierarchical architecture graph, which allows for simple compilation using codelets by representing connectivity of diverse architectures along with embedded metadata in the nodes. The embedded metadata describes architecture attributes including, but not limited to, storage node capacity, communication bandwidth, input and output ports, latency, and computation node capabilities which describe operations supported by the architecture and more.

Figure 3 below shows an example of an ADL program and visual representation of a HAG.



```
with hag.ComputeNode(name="Systolic Array") as sys_array:

        # components/nodes
        with hag.ComputeNode(name="PE Array") as pe_array:
                ...

        with hag.StorageNode(name="WBUF") as wbuf:
                ...

        sys_array.add_node(pe_array)
        sys_array.add_node(wbuf)
        ...

        # communication/edges
        sys_array.add_edge(wbuf, pe_array)
        ...
```

Figure 3: An example of an ADL program and visual representation of a HAG.

Compilation to Target Accelerators Having devised an abstract which is capable of representing different types of accelerator architectures, a multi-stage compilation process can be re-used across different HAGs.

The stages of compilation consist of the following:

- Operation mapping/scheduling, where a *mg*-DFG will be ordered to a sequence of operations, and each operation will be mapped to a particular component in the HAG. In addition, sequences of operations will be fused together according to user-supplied parameters.

- Compilation optimization, where a search for optimal compilation parameters will be performed using specifications of the HAG, including but not limited to, tiling sizes, loop unrolling factors, dataflow etc. During this process, data communication instructions/operations are necessarily added according to these parameters.
- Finally, the resulting instantiated capabilities are used to generate code for the target HAG.

Using an input lowered *mg*-DFG, the first phase which utilizes the HAG in the compilation pipeline is the mapping and scheduling of the *mg*-DFG operations to different HAG nodes.

As shown in Figure 4, the input *mg*-DFG will be structured as an ordered sequence of operations. To map the *mg*-DFG, the nodes will be iterated over and mapped to a certain HAG node according to the *mg*-DFG node operation and what sequence of capabilities are able to produce the equivalent operation. Once each of the nodes are mapped to the appropriate HAG nodes, they can be fused together to avoid excess off-chip memory accesses.

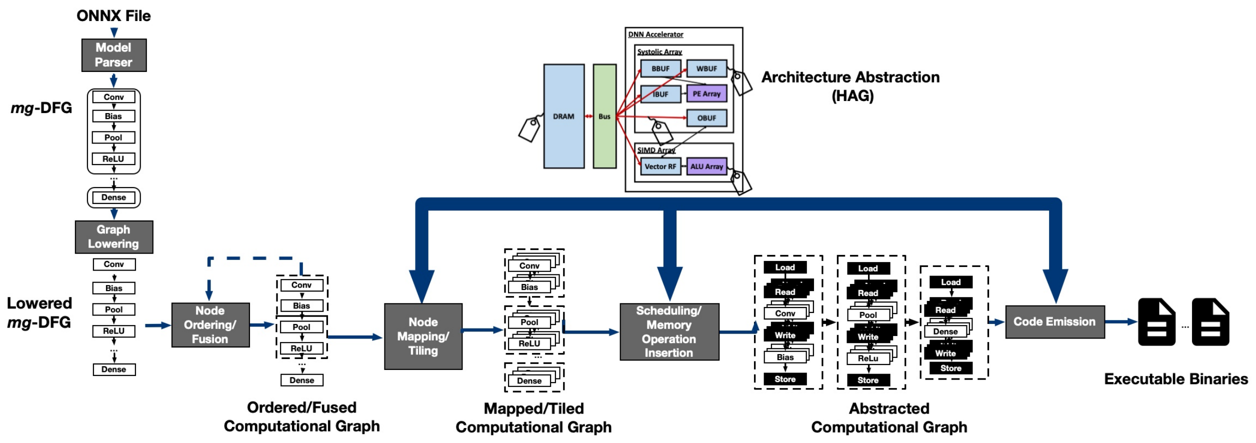


Figure 4: ONNX-to-hardware mapping flow through the *mg*-DFG and HAG representations.

Once the appropriate operations have been mapped, scheduled, and fused, the operations can be tiled and communication instructions can be inserted based on the attributes of the HAG nodes. Tiling operations consists of splitting the outputs of the last node of a fused sequence of operations into appropriately sized chunks of operations, mapping the outputs to the required inputs. This step also requires using the HAG nodes to determine the size of the tilings, as the input data is limited by the sizes of the on-chip memory used to store them. With a massive search space of ways to tile such operations, an iterative approach which uses reinforcement learning to determine the optimal tiling can be used in this step to find the best tiling size for each set of fused operations.

Once the appropriate tiling has been determined for each fused sequence, data communication operations must be added in a possibly cascading sequence of loops. Specifically, each tile corresponds to a read instruction from the on-chip HAG nodes, with sizes and bandwidth of the data communication constrained by the attributes of the node attributes. Off-chip communication operations are also required for loading in subsequent sets of tiles in cases where the on-chip memory is unable to store all of the necessary input data. Again, the HAG nodes aid in structuring the number and specification for each load instruction by specifying the bandwidth, size, latency, and other required attributes for successful compilation. Lastly, the corresponding write/store operations for each tile must be added to the sequence of operations in order for the next group of operations to successfully be executed.

As a final step, all operations generated using the HAG need to be compiled to valid code for the target accelerator. To perform this step, the finalized sequence of operations is iterated over, with each operation corresponding to a capability which stores a code template for the operation. Within the operations, the input and destination information is supplied to an *emit* function included in each capability, which then proceeds to generate the correct code based on the operation instance. As an example, a *read* operation which reads from the output buffer into the PE array generate code using the *read* capability in the systolic array, where the source and destination of the code is generated according to the *read* capability template.

PolyMath Compiler Integration

The aforementioned HAG is the integrated with the mg-DFG to enable flexible compilation. The integration of these two components of the VeriGOOD-ML framework forms a compiler which is flexibly capable of targeting varying types of accelerator, in addition to different configurations of a given accelerator. The compiler combines code templates called codelets with the mg-DFG node attributes and HAG attributes. Importantly, by using architecture attributes included in the HAG, the compiler can intelligently optimize and schedule the codelets for the accelerator. This workflow is illustrated in Figure 5. The remaining task for compilation is implementing the codelets for the different operations included in the mg-DFG. Fortunately, the codelet abstraction uses an intuitive framework built in Python for simple template implementations.

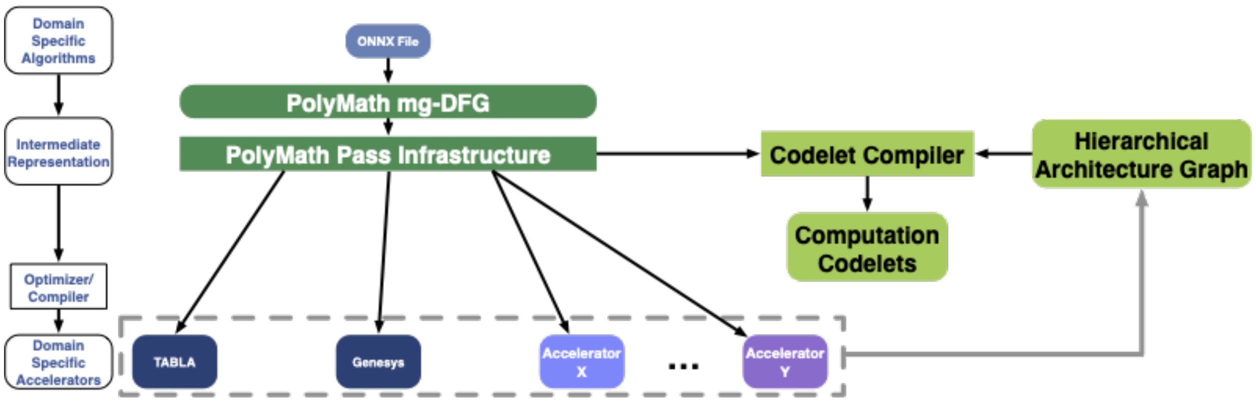

Domain Specific Algorithms
Intermediate Representation
Optimizer/ Compiler
Domain Specific Accelerators
ONNX File
PolyMath mg-DFG
PolyMath Pass Infrastructure
Codelet Compiler
Hierarchical Architecture Graph
Computation Codelets
TABLA
Genesys
Accelerator X
...
Accelerator Y

Figure 5: Compilation flow combining the HAG and mg-DFG to generate computation codelets for execution on different types of accelerator.

Computation Codelets To enable generation of executable binaries across a variety of different accelerators and accelerator configurations, a low-level abstraction is required which is capable of bridging the gap between the high-level intermediate representation of the mg-DFG and low-level instructions. We have devised a concept called codelets which represent instruction templates for target accelerators. The mg-DFG is converted to this same abstraction for every type of accelerator, with the only difference being the underlying instruction template used for binary generation. There are four primary types of codelet:

- Compute Codelet: Compute codelets represent instructions for performing computations on data.
- Memory Codelet: Memory codelets represent instructions which move a certain amount of data from one memory location to another. This can consist of both load/store to/from off-chip memory as well as to-from on-chip memory for read/writes.
- Loop Codelet: Loop codelets represent repetition of operations performed over a number of loop levels. They do not necessarily represent instructions unless explicit loop instructions exist in the target ISA. These can be used to parameterize other instructions.
- Control Codelet: Communication codelets represent instructions where some sort of control mechanism is used to determine whether instructions should be executed.

These codelets can be combined to form operations which match the semantics of execution for a given mg-DFG node.

Another key component of computation codelets is the underlying templates which are used for final generation of the binaries. The underlying templates consist of a sequence of valid instructions for the target accelerator, along with the required instructions fields/operands. The fields/operands are left blank or can be partially filled in when constructing the computation codelet representing an mg-DFG node.

Codelet Compiler and Optimizer The codelet compiler uses a combination of an input mg-DFG, HAG, and compute codelets to generate executable binaries. The process consists of a series of compiler passes which traverses the mg-DFG and constructs the compute codelets iteratively.

The first pass consists of traversing the nodes of the mg-DFG, and mapping the correct compute codelet to each of the nodes and creation of a logical memory map for all of the edges of the mg-DFG. Mapping compute codelets consists of identifying the type of mg-DFG node (e.g., 'conv2d') and mapping it to a manually constructed conv2d codelet. The mapped codelet will be added to the sequence of codelets, but will remain unparameterized until later compilation passes. In addition, a logical mapping of off-chip memory locations, called relocatables, for all of the data used in the mg-DFG is constructed for computing memory offsets throughout program execution. This is accomplished by using the datatype and size information specified by the mg-DFG node, and keeping track of previously mapped nodes. Later, this information can be used to compute memory locations when loading/storing data from the operations.

The second pass consists of adding constraints to the parameters of the mapped codelets based on the architectural information in the HAG. This can be accomplished by traversing the mg-DFG and the newly accumulated sequence of codelets, and identifying constraints such as SIMD array size or on-chip memory size, and using them to reduce the amount of possible values for SIMD operations performed in a single cycle, or the different sizes of tile for loads/stores.

The third step consists of selecting parameter values for the templates based on collected information from other codelets. Having constrained the number of values which can be used to parameterize the template, we can now fill out the fields for each of the instructions in the codelets. This quarter, we were able to generate the possible values and naively select template values based on the constraints.

Lastly, the templates will have been filled out and binaries can be emitted for compilation.