PyRTLMatrix: an Object-Oriented Hardware Design Pattern for Prototyping ML Accelerators

Dawit Aboye∗
UC Santa Barbara
dawit@ucsb.edu
Dylan Kupsh∗
UC Santa Barbara
dkupsh@ucsb.edu
Maggie Lim∗
UC Santa Barbara
maggie@ucsb.edu
Jacqueline Mai∗
UC Santa Barbara
jacquelinemai@ucsb.edu
Deeksha Dangwal
UC Santa Barbara
dee@cs.ucsb.edu
Diba Mirza
UC Santa Barbara
dimirza@cs.ucsb.edu
Timothy Sherwood
UC Santa Barbara
sherwood@cs.ucsb.edu

Abstract
As Machine Learning (ML) applications become pervasive and computer architects further integrate hardware support, the need to rapidly explore trade-offs between algorithms and hardware becomes pressing. While prior work on hardware accelerators has led to tremendous performance and energy improvements, it can be difficult to generalize these approaches without resorting to special purpose tools or even languages. Through object-oriented design principles we describe a general and reusable approach for generating parameterized neural network hardware. Specifically, we describe our experiences with high-level hardware design objects for building neural network hardware based on the open-source Python HDL, PyRTL. By thinking at a higher level of abstraction than simple "hardware modules," we open the door to a process by which hardware can be developed with software engineering principles. This creates new opportunities for a tight feedback loop between machine learning algorithm innovation and hardware design reality. Future works considering hardware development for ML applications can benefit from our work analyzing the costs and benefits of abstraction.

ACM Reference Format:

1 Introduction
Increasingly the field of computer architecture is turning to domain-specific acceleration to keep up with the user data and demand in machine learning [4, 9, 13]. Whether those accelerators are implemented as special purpose ASICs, through re-configurable hardware, or a combination of the two, there is a need for tighter coordination between the algorithms and architectures. Unfortunately, hardware design cycles take a significant amount of time and engineering effort, even with extensive expertise. The natural abstraction in hardware design is as static “modules” and the decomposition of designs into these modules is at odds with the cross-layer exploration required in this rapidly moving space. While high-level synthesis certainly has a role to play, current tools require both a great deal of developer hand holding (e.g. through the addition of pragmas) and understanding of how the compiler will interpret one’s C-code as hardware [15].

We explore an alternative approach where hardware description is explicitly under the control of the programmer (by directly instantiating components such as adders and memories), but at the same time standard software abstractions (such as functions, objects, and dictionaries) can be leveraged to manage the complexity of the hardware. Under such a model a natural question that arises is: how exactly should one structure software to best abstract the hardware it describes?

While there is no one single answer to this question, we explore a general and reusable approach for generating parameterized hardware specifically in the context of a neural network accelerator design. Building on PyRTL [6], a Python-based hardware design framework, we present PyRTLMatrix, a general purpose machine learning design pattern for the instantiation of neural network primitives. PyRTL allows the development of hardware to use the object-oriented structure of Python in interesting new ways while still preserving our ability to reason about the hardware described through execution. While these abstractions are helpful in encapsulating certain aspects of the complexity of the hardware design, other cross-cutting issues (not typically faced in software design) come to the fore. For example, the bitwidth of operations inferred locally (e.g. two 4-bit numbers add to a 5-bit number) and repeatedly can incrementally lead to very sub-optimal designs. We describe the development of PyRTLMatrix, including the ways in which the abstractions it provides are useful, understandable, and composable and the areas where unique use case of hardware development
presents new challenges for elaboration through execution. This paper presents the following contributions:

- We demonstrate the use of object-oriented design (OOD) principles to create high level hardware design patterns in PyRTL.
- We present the PyRTLMatrix class as a hardware design pattern for instantiating neural network accelerators.

Figure 1. Overview of our design flow. We transfer trained parameters from PyTorch to PyRTL, then compare the accuracy between the two networks.

2 Designing Hardware Pythonically

Traditional Hardware Description Languages (HDLs), such as Verilog have steep learning curves. PyRTL [6] is a Python-based HDL and its overarching goals are simplicity, usability, clarity, and extensibility. PyRTL includes classes for design, simulation, and testing. Unlike other HDLs, PyRTL emphasizes clarity rather than optimization by providing modularity and abstraction. By taking away some control from the user and replacing it with an intermediate structure that includes a complete tool chain, PyRTL gives the user a clear, high-level understanding of the creation, analysis, and testing of hardware.

3 The PyRTLMatrix class

The main operation in the inference step of a neural network is the matrix multiplication between the input vector and the matrix of weights in each layer. To intuitively create and perform matrix operations, we designed the PyRTLMatrix class. The PyRTLMatrix allows for the abstraction of hardware necessary to create a hardware representation of matrix operations. An instantiation of a PyRTLMatrix represents a wire bus that directs data through various logical computations representing matrix operations. The matrix multiply operation, displayed in Figure 2, implements an iterative solution utilizing base PyRTL multiplication algorithms as well as previously implemented base PyRTLMatrix functions. Creating hardware from an object-oriented perspective results in a simpler function design. Additionally, the PyRTLMatrix class implements the decorator pattern, a feature of the Python programming language, to ensure equal bitwidth across every element, as seen in Figure 3.

In Figure 4, we represent all currently supported base PyRTLMatrix functions with their equivalent mathematical expressions. Traditional object-oriented design patterns allow testing of each function individually and checking of correct results.

```python
def __matmul__(self, other):
    ''' Performs the matrix multiplication operation. 
    Is used with a @ b 
    :param PyRTLMatrix a: the first matrix. 
    :param PyRTLMatrix b: the second matrix. 
    :return: a PyRTL Matrix that contains the dot product of the two PyRTL Matrices. 
    '''
    result = PyRTLMatrix(self.rows, other.columns)
    for i in range(self.rows):
        for j in range(other.columns):
            for k in range(self.columns):
                result[i, j] = mult.
                    fused_multiply_adder(self[i, k],
                        other[k, j], result[i, j])
                result[i, j] = result[i, j][:32]
    result.bits = len(result[0, 0])
    return result
```

Figure 2. The implementation of the matrix multiply function. The matrix-multiply function demonstrates the relative simplicity of performing basic matrix operations in hardware.

```python
@property
def bits(self):
    ''' Gets the number of bits 
    :return: the number of bits.''
    return self._bits

@bits.setter
def bits(self, bits):
    ''' Sets the number of bits. 
    :param int bits: The number of bits.''
    self._bits = bits
    for i in range(self.rows):
        for j in range(self.columns):
            self[i, j].bitwidth = self._bits
```

Figure 3. The decorator pattern is used to ensure a consistent bitwidth among all items in the PyRTLMatrix.

Trade-offs arise from the implementation from utilizing the PyRTLMatrix class. Users are distanced from the hardware implementation and can replicate a software-like approach. In exchange for relative simplicity and ease of access, the abstraction can result in a loss of control and low-level understanding.

4 Evaluation

Our pipeline consists of two main components: the software network in PyTorch [14] and the hardware implementation.
Our PyTorch model has three main parts: a layer of weights, an element-wise activation function, and an argmax function for the final classification. We train our software neural network with higher parallelization of matrix operations, despite requiring the same number of hardware units. This means that the exact same number of operations is being performed on a single clock cycle between all the different “blocked” designs. The same number of fused multiply-adders (FMAs) are built (blocking only reduces the number of operations performed in sequence rather than in parallel), meaning, we still perform the same number of multiplication operations. We analyze why area and frequency improve of operations performed in sequence rather than in parallel), meaning, we still perform the same number of multiplication operations. We analyze why area and frequency improve of operations performed in sequence rather than in parallel), meaning, we still perform the same number of multiplication operations. We analyze why area and frequency improve of operations performed in sequence rather than in parallel), meaning, we still perform the same number of multiplication operations. We analyze why area and frequency improve of operations performed in sequence rather than in parallel), meaning, we still perform the same number of multiplication operations. We analyze why area and frequency improve.
where

\[ n \] wires grow over a series of accumulations in a Kogge-Stone adder. There is no inherent sequential logic—no unit is automatically reused once it is instantiated, which can result in sprawling data being passed until every iteration is replicated in hardware. As the experiment demonstrates, this is achieved by creating as many hardware units as necessary through which the data flows. Using the same set of nested loops (as shown in Figure 2), we can build a hardware design replicating the software counterpart.

PyRTLMatrix is a domain specific language and compiler that translates high level code to hardware. Richmond et al., for example, bring the power of higher order functions [15] to hardware abstraction. Working at this high level of abstraction removes complexity of hardware construction, since we can simply plug in an existing hardware unit for a particular operation wherever we need it, such as in the PyRTLMatrix.

Pulling out these design patterns into discrete modules saves time and is intuitive to reason about from a software perspective while designing the hardware neural network.

However, this increased convenience comes at the cost of the hardware sophistication. In software, matrix multiplication is typically an iterative function requiring nested loops. Using the built-in matrix multiply of PyRTLMatrix, we can build a hardware design replicating the software counterpart, using the same set of nested loops (as shown in Figure 2). As the experiment demonstrates, this is achieved by creating as many hardware units as necessary through which the data is passed until every iteration is replicated in hardware. There is no inherent sequential logic—no unit is automatically reused once it is instantiated, which can result in sprawling and unoptimized designs. However, PyRTL still allows us to delve deeper into each abstraction and reason about trade-offs. A high-level abstraction like PyRTLMatrix allows for rapid prototyping with the opportunity for advanced designs provided that users properly analyze the design at the circuit level.

Figure 6. Block size vs. logic area, max. clock frequency. As we use wider matrix blocks for matrix multiplication, the logic area increases and the clock frequency declines. We generate a set of multipliers to be executed in parallel. A smaller multiplier requires a shorter accumulation over the width of the matrix, limiting bitwidth and area, and allowing a faster overall process time per image. From these trends, we see this increased parallelization improves performance despite the fact that the same number of operations must still be performed regardless of block size.

Figure 7 demonstrates how the bitwidth of input and output wires grow over a series of accumulations in a Kogge-Stone adder, the hardware adder used in the FMA. Specifically, the Kogge-Stone adder grows in area at a rate of \( O(n \log n) \), where \( n \) is the bitwidth of the largest input wire. A longer row results in increasingly larger adders across the row. Block multiplication avoids longer accumulations, which limits the bitwidth growth of the adder inputs and, by extension, the area of the neural network as a whole. This iterative tree generation is a specific example of a cross-cutting issue that would not normally be encountered in either typical software or hardware design. This intersection of hardware and software practices has benefits and drawbacks.

Working at this high level of abstraction removes complexity of hardware construction, since we can simply plug in an existing hardware unit for a particular operation wherever we need it, such as in the PyRTLMatrix. Pulling out these design patterns into discrete modules saves time and is intuitive to reason about from a software perspective while designing the hardware neural network.

However, this increased convenience comes at the cost of the hardware sophistication. In software, matrix multiplication is typically an iterative function requiring nested loops. Using the built-in matrix multiply of PyRTLMatrix, we can build a hardware design replicating the software counterpart, using the same set of nested loops (as shown in Figure 2). As the experiment demonstrates, this is achieved by creating as many hardware units as necessary through which the data is passed until every iteration is replicated in hardware. There is no inherent sequential logic—no unit is automatically reused once it is instantiated, which can result in sprawling and unoptimized designs. However, PyRTL still allows us to delve deeper into each abstraction and reason about trade-offs. A high-level abstraction like PyRTLMatrix allows for rapid prototyping with the opportunity for advanced designs provided that users properly analyze the design at the circuit level.

5 Related Work

Recently, there has been a tremendous growth in machine learning acceleration [4, 5, 9] to keep up with the ever-increasing demand for better performance. We do not present a machine learning accelerator, but instead present hardware design patterns to accelerate the prototyping and evaluation of accelerators. We aim to improve the accessibility of commonly used hardware design patterns in ML accelerators.

High level synthesis (HLS) tools also allow developers to abstract away complexities of hardware design to convert high level code to hardware. Richmond et al., for example, bring the power of higher order functions [15] to hardware design using C/C++. It avoids the many difficulties associated with using traditional HDLs to design hardware, such as low level programming, registers, scheduling, by utilizing a syntax similar to modern software languages to provide a higher level abstraction of hardware design. Similarly, SPATIAL [10] is a domain specific language and compiler that also allows for more productivity using software concepts like nested loops, while still allowing access to lower level concepts like memory hierarchy and memory transfers explicitly. There are other HLS tools like TABLA [13] that gear specifically towards machine learning and act accelerator generators for machine learning algorithms. However, unlike TABLA, our work focuses more on a design flow, rather than automatically generating hardware based on a specified learning model.

Traditional HDLs like Verilog prove difficult to learn and use, especially for software engineers, and lack the modularity that allow reuse of components. Therefore other tools
that allow developers to design hardware using more familiar software-like interfaces have been developed. For our work, we use PyRTL, but other projects would provide similar functionality. Chisel [2] is an advanced hardware design language embedded in Scala. Similar to PyRTL, it is object-oriented and has type inference to further abstract from details of hardware design. CAsH [1] and Lava [3], are Haskell-based functional HDLs the provide features like simulation, formal verification and generation of code for implementable circuits. MyHDL [7] uses Python generators and decorators, looks similar to Verilog, and comes with a wide variety of features like synthesis, simulation, test bench creation, and optimization. PyMTL [12] and Mamba [8], also Python-based, provide a vertically integrated tool for functional-level, cycle-level and register-transfer level modeling.

While traditional HDLs lack the abstraction and modularity needed to create our reusable neural network functions, HDLs mentioned here do support hardware design abstraction. We built our neural network design pattern using PyRTL and the implementation of foundation classes, such as the PyRTLMatrix class abstracted away the low-level details. This allowed our process to behave more like HLS while still giving us control over writing and reusing of low-level hardware design patterns.

6 Conclusion

As compute-intensive workloads such as machine learning grow in popularity, it is becoming clear that application specific accelerators hardware are here to stay. Designing algorithms and programming them quickly becomes key when trying to keep up with new and emerging applications. In this work, we make the position that there is a need to accelerate the process of designing accelerators, and we do this using hardware design patterns. PyRTLMatrix class is based on object-oriented design principles and provides reusable design primitives to quickly instantiate neural network inference accelerators. The functions of this class concisely describe neural network operations in hardware and facilitate rapid prototyping of new algorithms and associated design trade offs. Looking ahead, hardware design patterns and templates would be the first step toward automatic synthesis of machine learning accelerators. Future studies should look to create automatic translation of PyTorch neural networks into HDLs like PyRTL.

References


