# 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
maggielim@ucsb.edu

Jacqueline Mai[*]
UC Santa Barbara
jacquelinemai@ucsb.edu

Deeksha Dangwal
UC Santa Barbara
deeksha@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:**
Dawit Aboye, Dylan Kupsh, Maggie Lim, Jacqueline Mai, Deeksha Dangwal, Diba Mirza, and Timothy Sherwood. 2019. PyRTLMatrix: an Object-Oriented Hardware Design Pattern for Prototyping ML Accelerators. In *Proceedings of The 4th Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2'19).* ACM, New York, NY, USA, 5 pages.

## 1 Introduction

Increasingly the field of computer architecture is turning to domain-specific acceleration to keep up with 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:

[*]Authors were a part of an undergraduate research team.

*EMC2'19, June 23rd, 2019, Phoenix, AZ, USA*
2019.

```
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.

- 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.

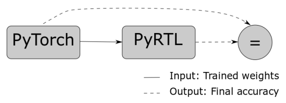

**Figure 1.** Overview of our design flow. We transfer trained parameters from PyTorch to PyRTL, then compare the accuracy between the two networks.

## 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.

```
@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`.

## 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. 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 in PyRTL, with shared trained parameters, as seen in Figure 1. Our all-Python setup allows for intuitive translation between the two networks. We use this setup to classify hand-written numbers using the MNIST dataset [11].

| Operation | PyRTL Matrix |
|---|---|
| A@B<br>Matrix Multiply | __matmul__(self, B) |
| A * B<br>Element-Wise Multiply | __mul__(self, B) |
| A + B<br>Element-Wise Add | __add__(self, B) |
| $A^T$<br>Transpose | transpose(self, T) |
| x = A[k]<br>Get Element | __getitem__(self, k) |
| A[k] = v<br>Set Element | __setitem__(self, k, v) |

**Figure 4.** An overview of the basic operations in the PyRTLMatrix class. The functions take advantage of Python's built-in "magic methods" for more intuitive understanding. Abstracting hardware into simplified operations allows for code reuse. Additionally, the functions make test creation and error isolation easy.

PyTorch Forward Function

```
def forward(self, x):
  out = x.view(BATCH_SIZE, -1)
  for i in range(len(self.weights)):
    out = self.weights[i](out)
    out = self.relu(out, threshold)
  return out
```

PyRTL Forward Function

```
def forward(self, x):
  out = x
  for i in range(len(self.weights)):
    out = self.weights[i].toMatrix()@out
    out = out + self.bias[i].toMatrix()
    out = self.relu(out, int(threshold))
  return matrix.argmax(out)
```

**Figure 5.** The forward function implemented within PyTorch and PyRTL. Using the PyRTLMatrix class, the PyRTL code becomes simpler despite integrating hardware designs. Through the application of software design patterns, hardware design languages become easier to understand.

Our PyTorch model has three main parts: a layer of weights, a non-linear activation function, and an argmax function for the final classification. We train our software neural network and collect the trained weights and biases. We maintain positive weights and input vectors clamped between [0,1] during training and quantize them to ensure 8-bit integers are used in the PyRTL design. We perform inference in PyTorch with the forward function, as shown in the first row of Figure 5.

We implement an equivalent forward function in hardware, using the PyRTLMatrix as our basic primitive. When the design is simulated, the trained weights and biases for each layer are first loaded from file into on-circuit memory before the logic for the forward function is elaborated. This implementation of the forward function in PyRTL is shown in the second row of Figure 5, which shows weights, biases, and the input vectors represented as PyRTLMatrix objects.

### 4.1 Case study

Our block matrix algorithm splits a single large multiplication into a series of smaller operations and accumulates the final result. This is a variation to the PyRTL forward function presented in Figure 5. Although these implementations are functionally equivalent, blocking allows us to examine latency-area trade-offs. For example, our implementation uses 16 smaller 10x49 block matrix multiplication blocks to perform a single 784x10 matrix multiply. This experiment also gives us the opportunity to observe the effects of increased abstraction on hardware design.

**I. Experiment:** We first pad the 10-by-784 layer in the network to be 800 columns across, to get more standard block sizes, and then choose a list of factor pairs to determine several block sizes with which to test the network design: 10 columns (80 multipliers), 16 columns (50 multipliers), 25 columns (32 multipliers), 50 columns (16 multipliers), and 160 columns (5 multipliers). Working with a single layer feedforward neural network design, we record performance metrics of our design for different block sizes and plot the relationship between multiplier blocking, design area, and maximum clock frequency as seen in Figure 6. Measurements for area and clock frequency are made with the area_estimation and TimingAnalysis functions included in the PyRTL analysis module. A 130*nm* technode is assumed for all area measurements.

**II. Analysis:** As shown in Figure 6, the overall trends of the experimental data indicate that a smaller block multiply results in smaller logic area and a faster maximum clock frequency. Area is plotted on the left vertical axis and max. frequency on the right axis. It is important to note that, due to the iterative nature of the block matrix algorithm in the design, one image is processed fully on a single clock cycle.

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 with higher parallelization of matrix operations, despite requiring the same number of hardware units.

Using block matrix multiplication results in a tree-like structure in hardware where each block multiplication is performed in parallel. The resulting WireVectors are added together in a cascading series of element-wise vector adders. Figure 7 demonstrates how the bitwidth of input and output wires grow over a series of accumulations in a Kogge-Stone

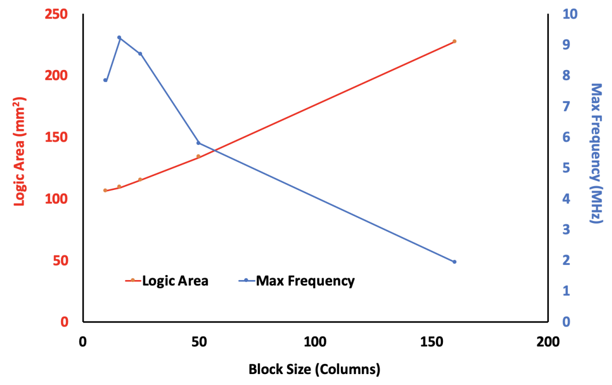

**Figure 6.** Block size vs. logic area, max. clock frequency. As we use wider matrix blocks for matrix multiplication, the logic area increases and the clock frequency declines. We generate a set of multipliers to be executed in parallel. A smaller multiplier requires a shorter accumulation over the 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.

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 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 tradeoffs. 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.

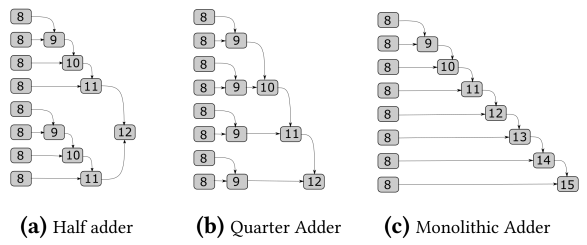

**Figure 7.** An overview of how bitwidth grows with the depth of an adder tree. Each shaded cell indicates the bitwidth of a `WireVector`. The connections indicate when a wire is being added to another to yield a third, wider, wire. Here, an accumulation across a single row of 8 elements in shown monolithically and broken down into blocks two different ways. The two blocked additions, **(a)** and **(b)**, have smaller final bitwidths and shallower critical paths than the monolithic operation, **(c)**.

## 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. C$\lambda$aSH [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 and Future Work

As compute-intensive workloads such as machine learning grow in popularity, it is becoming clear that application specific hardware accelerators 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 would look to create automatic translation of PyTorch neural networks into HDLs like PyRTL. We would also add more functions to the `PyRTLMatrix` class to support convolution and pooling operations.

## References

[1] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. C$\lambda$aSH: Structural descriptions of synchronous hardware using haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 714–721.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 174–184.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.

[5] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[6] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–7. https://doi.org/10.23919/FPL.2017.8056860

[7] Jan Decaluwe. 2004. MyHDL: a Python-Based Hardware Description Language. *Linux journal* 127 (2004), 84–87.

[8] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. 2018. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[9] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246

[10] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 296–311.

[11] Y. LECUN. [n. d.]. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/* ([n. d.]). https://ci.nii.ac.jp/naid/10027939599/en/

[12] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A unified framework for vertically integrated computer architecture research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 280–292.

[13] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 14–26.

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

[15] Dustin Richmond, Alric Althoff, and Ryan Kastner. 2018. Synthesizable Higher-Order Functions for C++. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2835–2844.