2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Verification of Accelerators in System Context

Russell A. Klein
Mentor, A Siemens Company
8005 SW Boeckman Rd
Wilsonville, OR 97070

Abstract - **Hardware can be used to accelerate operations on a processor, by taking some of the processing load originally handled by software and moving it into hardware as a dedicated accelerator. These accelerators need to be verified in the traditional manner, but will also benefit from being verified in the context of the larger body of software in which they run. This paper explores the development and verification of a TensorFlow accelerator used in a simple machine learning (ML) example. TensorFlow graphs are typically written in Python. They are generally executed in Python, with input and output data streams being pre- and post-processed in Python. We will describe how to verify the accelerator in the context of the execution of the complete TensorFlow graph in Python. This will be shown with Python running on a host computer, a virtual machine modeling an embedded processor/memory sub-system, and a full RTL system environment. While this will be shown for a TensorFlow/Python example, the techniques shown are generally applicable where accelerators are used in a software context.**

I. INTRODUCTION

For many electronic systems implemented in silicon as ASICs, there will be no "next node". The economics of the most advanced silicon geometries dictate that they will be used only for the highest volume devices. This has some interesting implications for embedded system design. In the past, designs without sufficient processing power to deploy an algorithm in software needed only to wait a while. In a short time, ARM would come out with a new more powerful processor, and TSMC would unveil a silicon process on a smaller geometry node delivering higher performance and lower power. With no effort or expense a more capable computing platform could be employed. Without the benefit of migration to smaller nodes, "doing nothing" is no longer a viable strategy. Developers will need to find creative ways to address the gap between available processing power and system requirements. One approach is to develop accelerators that move functions from software into bespoke hardware implementations that deliver higher performance at lower power.

Often, algorithms used in embedded systems are initially developed on desktop compute environments. As they are moved to embedded systems, developers would like to minimize the changes made. This is especially true for complex algorithms like the latest generation of machine learning systems. However, hardware and software are vastly different in implementation. As a result, the description (the source code) will likely go through significant changes as implementation proceeds. Verification of the implementation cannot wait until the end of the development cycle. Waiting until the final implementation to verify the correctness of even moderately complex algorithms can result in a nearly intractable debugging problem. One needs to continuously verify that the algorithm remains intact and correct as it migrates from its original to its final implementation, and all stops in between as different architectural alternatives are considered.

II. VERIFICATION OF ACCELERATORS

The traditional techniques of verification, UVM, directed tests, constrained random, coverage metrics, etc., can and should be applied to the verification of accelerators during their development. However, as developing an accelerator involves moving a function from software to hardware the verification should also prove that the accelerator works in the context of the larger body of software from which it originated,

and delivers the same results given the same stimuli. To do this, one needs to interface the new implementations of the algorithm into the larger system and exercise it with a common set of inputs. Typically, the original execution environment could be used to determine the expected results, and comparison of expected versus actual results can be performed in that context.

A. Overview

An accelerator performs a function that would normally done in software, but due to performance or power constraints is moved into an alternate, more efficient, implementation. For the purposes of this paper we shall assume that the implementation will be ASIC or FPGA, through synthesizable RTL. The first step of verification is to prove the original implementation is working correctly. Then a behavioral implementation is created that incorporates interfacing that is compatible with a hardware implementation. This usually involves modifying the interface so that all parameters, input and output, are passed explicitly. Accommodations are made for passing data by reference, and any out of scope (global) references are removed. The new function, *f_prime()*, could be called from the original program with the same inputs as a concurrent call to *f()*. The outputs of *f()* are used as an expected result for *f_prime()*.

The accelerator is successively refined to ultimately reach a full RTL implementation. Each step along the way the same approach described above can be used to verify consistency between the original algorithm and the resulting implementation. The next step would be to optimize the algorithm for data flow and register widths. Then there would be a conversion to RTL for the accelerator. Next the host based software execution environment would be replaced with a virtual machine (VM) of the target system. Finally a complete RTL system will be realized. With each successive refinement a similar verification approach can be done. As the model becomes more detailed smaller verification datasets would be used to keep the verification time tractable.

To illustrate how this verification of accelerators will take place in system context, we will walk through the verification of a simple machine learning algorithm implemented in TensorFlow. TensorFlow is a commonly used open source machine-learning platform [1]. As a simplification, we will be looking at just the inferencing side of this algorithm, which is significantly simpler than the corresponding training algorithm. TensorFlow machine learning programs that employ the gradient descent algorithm will have both forward flow code as well as back propagation code. The back propagation code records the values seen in the forward pass and computes the local first order partial derivative for the operation. This derivative is used to perform adjustments to the convolution kernels during training sessions. For inferencing applications, this code can be omitted.

We will use the Yolo-tiny [2] algorithm as an example for verifying accelerators in a system context, specifically Yolo-tiny v3. Yolo-tiny is an object detection algorithm based on convolutional neural networks (CNNs). It implements a small neural network that is typically used on computationally constrained platforms. This is an algorithm that is high in computational complexity. The general principles described here are not limited to CNNs, but can be used for the verification of any function that is moved to hardware in the form of an accelerator. The original Yolo-tiny was implemented in Darknet can be found here: https://pjreddie.com/darknet/yolo. This was implemented in C. An example of a TensorFlow implementation is found here: https://github.com/gliese581gg/YOLO_TensorFlow. This is the implementation that is used for the examples in this paper. Note that it differs slightly from the original C version developed by J. Redmon.

B. Overview

Most machine learning algorithms will be developed in a desk-top or cloud environment. Typically they are run in one of the many popular machine-learning platforms. These are most commonly written in Python or other languages with good support for matrix processing. Yolo-tiny can be run on the desktop. While the profiling capabilities in TensorFlow are weak, the total computational load can be discovered as well as the distribution through the algorithm, giving hints as to where computational or dataflow bottlenecks might exist. Yolo-tiny, like many TensorFlow applications, is built in layers. For each of the computationally heavy layers we will create C equivalent descriptions, although they could be written in SystemC or SystemVerilog. This implementation will be run concurrently with the original TensorFlow to verify it in context. The C is then optimized and converted to RTL for implementation. The RTL is then

connected to the same environment for an equivalent verification. Next RTL is connected to an embedded platform, specifically a virtual prototype. This verifies that the transition from desktop environment to embedded system does not introduce any changes or errors. Finally, a full RTL implementation is performed; in this context overall system performance and power can be measured.

C. Desktop

Running Yolo-tiny on the desktop in Python and TensorFlow establishes a benchmark for correct operation and overall performance and dataflow. See table #1 for a list of the computations performed across the various layers of the CNN. From this we can observe that almost 99% of the computational load is embodied in a 3x3 2D convolution, followed by a 2x2 max pooling operation. There are 3.8 million feature values, and 25 million convolution filter values that will need to be moved to and from memory, or between computational units as the algorithm is run. The overall computational load is dominated by the 2.2 billion floating point multiplications and 2 billion floating point addition operations. But the movement of data will likely be the limiting factor in overall throughput. Note that performance on the desktop may not correlate to performance on the embedded system, as instruction level parallelism, caching, and memory access patterns may differ substantially between the desktop environment and the embedded target system. The data-flows, however, will remain constant.

It is beyond the scope of this paper to detail methods for determining the correctness of the algorithm at this point, or how to extract any profile information from the execution of the code. We shall proceed assuming the TensorFlow implementation is correct.

Layer	Type	Width	W * H	Filters	Channels	Feature size	Coeff size	Multiplies	Accumulates	Compares	Load %
1	3x3 conv	416	173056	16	3	519168	432	74,760,192	66,453,504		3.32%
2	2x2 max pool	416	173056		3	519168				389,376	
3	3x3 conv	208	43264	32	16	692224	4608	199,360,512	177,209,344		8.84%
4	2x2 max pool	208	43264		16	692224				519,168	
5	3x3 conv	104	10816	64	32	346112	18432	199,360,512	177,209,344		8.84%
6	2x2 max pool	104	10816		32	346112				259,584	
7	3x3 conv	52	2704	128	64	173056	73728	199,360,512	177,209,344		8.84%
8	2x2 max pool	52	2704		64	173056				129,792	
9	3x3 conv	26	676	256	128	86528	294912	199,360,512	177,209,344		8.84%
10	2x2 max pool	26	676		128	86528				64,896	
11	3x3 conv	13	169	512	256	43264	1179648	199,360,512	177,209,344		8.84%
12	2x2 max pool	13	169		256	43264				32,448	
13	3x3 conv	7	49	1024	512	25088	4718592	231,211,008	205,520,896		10.25%
14	3x3 conv	7	49	1024	1024	50176	9437184	462,422,016	411,041,792		20.51%
15	3x3 conv	7	49	1024	1024	50176	9437184	462,422,016	411,041,792		20.51%
16	matmul							12,845,056	11,417,828		0.57%
17	matmul							8,388,608	7,456,540		0.37%
19	matmul							6,021,120	5,352,107		0.27%
					Totals:	3,846,144	25,164,720	2,254,872,576	2,004,331,179	1,395,264	100%

Table #1 – Yolo-tiny layers and computations

A 3x3 2D convolution is an operation that is used commonly in machine learning and video processing. It multiplies a 3x3 array of numbers, element wise, with all 3x3 regions of an input array and sums the products to produce a value in the output array. Figure #1 concisely diagrams algorithm. While quite simple, the algorithm has a large variety of possible implementations. It is interesting to note that the algorithm is embarrassingly parallel. Within each layer all of the multiplications can be performed in concurrently, and all the additions can be performed in an adder tree no more than 4 layers deep. In software one could parallelize this across multiple CPUs. It can also be easily parallelized across a GPU, or TPU/NPU.

$$z(n_1, n_2) = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} x(k_1, k_2) y(n_1 - k_1, n_2 - k_2)$$

Equation #1 – 2D convolution formula

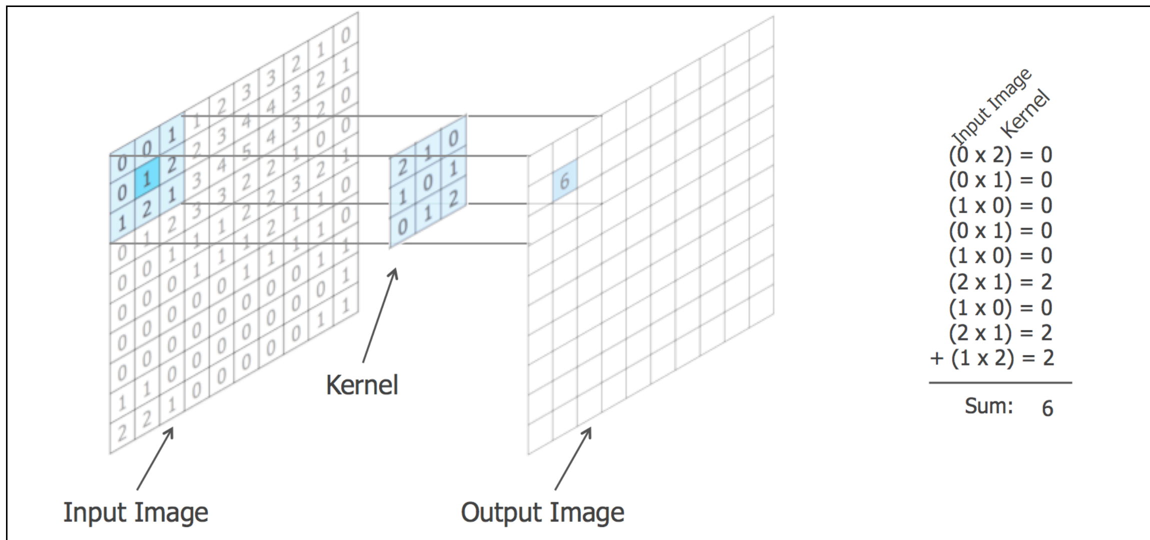

Figure #1 Convolution Algorithm

D. Algorithmic C

Our first step is to create the abstract implementation of the algorithm of the TensorFlow kernel to be accelerated. This algorithmic code can then be plugged in to TensorFlow as a user-defined kernel. This is run and verified on the desktop environment. The performance is such that large datasets can be run through the algorithm very quickly (relative to RTL implementations). This will prove the algorithm, as written, produces the same behavior as the original TensorFlow graph. Code coverage at this stage for the algorithm would be a desirable thing. During this run the inputs to and outputs from the kernel can be collected and saved as a database of stimuli and expected outputs for later verification.

```
1  void conv_2d(double *image, int size, double *kernel, double *image_out)
2  {
3      int i, j, r, c, x, y, idx;
4      double sum;
5
6      for (i=0; i<size; i++) {
7          for (j=0; j<size; j++) {
8              sum = 0.0;
9              for (r = 0; r<3; r++) {
10                 for (c = 0; c<3; c++) {
11                     x = i + r - 1;
12                     y = j + c - 1;
13                     idx = y * size + x;
14                     if ((-1<x) && (x<size) && (-1<y) && (y<size))
15                         sum += image[idx] * kernel[r*3 + c];
16                 }
17             }
18             image_out[j * size + i] = sum;
19         }
20     }
21 }
22
```

Listing #1 – 2D convolution algorithm

Listing #1 shows the algorithm for a 3x3 2D convolution function written in standard C. Those familiar with this algorithm should take note that the stride and size parameters are omitted from this implementation, as they are constant in this application. The function takes as input a square `image` of a given `size`, and a `kernel`, which is assumed to be a 3x3 array of floating point numbers. It returns a convolved image through the return parameter `image_out`. Lines 6 and 7 are the outer loop that goes over the input `image`. Lines 9 and 10 loop over the `kernel`. Lines 11 through 15 compute the product for each pixel in the image multiplied by each value in the kernel, on an element wise basis. The sum of these products is the resulting pixel for the output image.

The conditional expression on line 14 handles the cases where the pixel being referenced is outside of the bounds of the image (the "edge" condition). Pixels out of range of the image are assigned the value 0, and 0 multiplied by anything is 0. Thus a 0 would be added to the sum being tallied on line 15. Since adding a 0 to a quantity has no impact, neither the multiplication nor the addition are performed. In software this optimization is insignificant, but in a hardware implementation it may not be.

In the image recognition algorithm, the output of this convolution is fed into a 2x2 max pool computation. The max pool computation takes a 2x2 array of pixels and returns the largest of the 4. For brevity we omit the listing and details on this algorithm, as it is quite simple.

With the 3x3 convolution and 2x2 max pooling implemented in C, it needs to be verified to produce the same results as the original TensorFlow implementation. One way to do this is to call this C implementation from the Tensorflow graph. In this way it is run concurrently with the original TensorFlow layers, see figure #2. And the outputs of the Tensorflow layers and the C are compared (this is trivial in python) to check consistency.

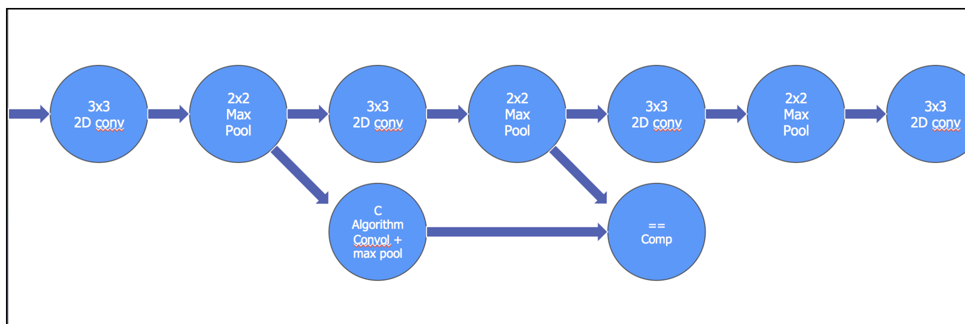

Figure #2 – C algorithm running in parallel with original Python

Listing #2 shows how the C code can be called from TensorFlow. It uses the TensorFlow user-defined operation. It creates a new operation, which can be called from the TensorFlow graph. It is passed Tensors (multi-dimensional arrays) as inputs and returns Tensors as outputs. These are handled by the TensorFlow code.

Lines 7 to 14 define the new operation. In this case, the new operator is called "CatapultConvMaxPool", and the input and output parameters are defined. Lines 20 to 38 define the class that is called when there is input data for the operation. Lines 25 to 32 get pointers and dimensions of the inputs. Line 36 makes the call to the C implementation. Line 40 calls a C pre-processor macro that registers the new operator with TensorFlow. The class is compiled into a shared library using a makefile provided by TensorFlow.

Listing #3 shows how this new operator is called from a Python TensorFlow program. In the `__init__` method there is a call to load in the shared library that contains the new operator. In the `build_network` method, the new operator is called. Note that it performs the same function as conv_layer 3 and pooling_layer 4. It takes as input "`pool_2`" and emits as output "`pool_4_c`". Observe that when sess.run is called, it calls for `pool_4` (the TensorFlow output) and pool_4_c (output from the equivalent C implementation) to be evaluated. These can be compared to verify that the C implementation matches the original TensorFlow.

Note that it is a TensorFlow convention to convert the CamelCase representation of the operator name to a lower case/underscore format when referenced as a python method, thus conv_max_pool is called when building the graph.

```
1  #include "tensorflow/core/framework/op.h"
2  #include "tensorflow/core/framework/shape_inference.h"
3  #include <stdio.h>
4  
5  using namespace tensorflow;
6  
7  REGISTER_OP("CatapultConvMaxPool")
8  .Input("cat_image: float")
9  .Input("cat_filter: float")
10 .Output("cat_image_out: float")
11 .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
12     c->set_output(0, c->input(0));
13     return Status::OK();
14 });
15 
16 #include "tensorflow/core/framework/op_kernel.h"
17 
18 using namespace tensorflow;
19 
20 class CatapultConvMaxPoolOp : public OpKernel {
21 public:
22     explicit CatapultConvMaxPoolOp(OpKernelConstruction* context) : OpKernel(context) {}
23 
24     void Compute(OpKernelContext* context) override {
25         const Tensor& input_tensor_image = context->input(0);
26         auto input_image = input_tensor_image.flat<float>();
27 
28         const Tensor& input_tensor_filter = context->input(1);
29         auto input_filter = input_tensor_filter.flat<float>();
30 
31         Tensor* output_tensor = NULL;
32         OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor_image.shape(),
33                                                          &output_tensor));
34         auto output_image = output_tensor->flat<float>();
35 
36         c_version_conv_max_pool(&input_image, &input_filter, &output_image)
37     }
38 };
39 
40 REGISTER_KERNEL_BUILDER(Name("CatapultConvMaxPool").Device(DEVICE_CPU), CatapultConvMaxPoolOp);
41 
42 
43 
```

Listing #2 – definition of a user defined TensorFlow kernel

```
def __init__(self,argvs = []):

        catapult = tf.load_op_library('./catapult_conv_max_pool_algorithm.so')

        self.argv_parser(argvs)
        self.build_networks()
        print("Self.fromfile: ", self.fromfile)
        if self.fromfile is not None: self.detect_from_file(self.fromfile)


def build_networks(self):
        if self.disp_console : print("Building YOLO_tiny graph...")
        self.x = tf.placeholder('float32',[None,448,448,3])
        self.conv_1 = self.conv_layer(1,self.x,16,3,1)
        self.pool_2 = self.pooling_layer(2,self.conv_1,2,2)
        self.conv_3 = self.conv_layer(3,self.pool_2,32,3,1)
        self.pool_4 = self.pooling_layer(4,self.conv_3,2,2)

        self.pool_4_c = catapult.conv_max_pool(self.pool_2, 16)

        self.conv_5 = self.conv_layer(5,self.pool_4,64,3,1)
        self.pool_6 = self.pooling_layer(6,self.conv_5,2,2)
        self.conv_7 = self.conv_layer(7,self.pool_6,128,3,1)
        self.pool_8 = self.pooling_layer(8,self.conv_7,2,2)
        self.conv_9 = self.conv_layer(9,self.pool_8,256,3,1)
```

```
def detect_from_cvmat(self,img):
        s = time.time()
        self.h_img,self.w_img,_ = img.shape
        img_resized = cv2.resize(img, (448, 448))
        img_RGB = cv2.cvtColor(img_resized,cv2.COLOR_BGR2RGB)
        img_resized_np = np.asarray( img_RGB )
        inputs = np.zeros((1,448,448,3),dtype='float32')
        inputs[0] = (img_resized_np/255.0)*2.0-1.0
        in_dict = {self.x: inputs}

        #net_output = self.sess.run(self.fc_19,feed_dict=in_dict)
        net_output, step4_python, step4_c = self.sess.run(self.fc_19, self.pool4, self.pool4_c, feed_dict=in_dict)
        compare(step4_python, step4_c)

        self.result = self.interpret_output(net_output[0])
        self.show_results(img,self.result)
        strtime = str(time.time()-s)
        if self.disp_console : print('Elapsed time : ' + strtime + ' secs' + '\n')
```

Listing #3 – instantiation of C algorithm of convolution and max pooling

Rather than using the python matrix equality operator we call a Python function that performs the comparison that allows for slight rounding errors in the floating-point operations. Getting the exact floating-point results to match the desktop TensorFlow calculation is unlikely and not even desirable.

This step can be repeated for each of the convolution/max pooling layers in the graph. Replace and verify each kernel individually. Once each kernel has been proven, then we start verifying appropriate combinations.

E. Bit Accurate C

This is the stage where optimizations of the implementation should be introduced. Running an inference for this implementation of Yolo-tiny on a MacBook Pro with a 2.9 GHz Intel i7 processor takes about 1.4 seconds. Adding one user-defined kernel into the graph increases that execution time to just under 5 seconds. Once the algorithm is moved to RTL it will take about 389 minutes to execute on a software based logic simulator, or about 5,000 times slower. Observing the impact of any changes to the algorithm will require inferencing across a large set of images. For example, if we have a library of 1000 images used to verify the correctness of the TensorFlow graph, it would take less than an hour and a half to run it in algorithmic C. But with just the accelerator in RTL the same verification would take 270 days.

Optimizations at this stage of the design process would involve things like converting the algorithm from using 32 bit floating point numbers to using a fixed point representation. This involves determining if smaller numeric representations than 32-bits would produce acceptable results. We accomplish this by converting the C code from using the native C types to bit accurate data-types and operations. This can be

done using the open source algorithmic C data types and class library available at http://hlslibs.org. Floating-point multipliers are more expensive in terms of area and power than fixed-point operations. As the size of the operands to a multiplier are reduced, the area and power are reduced at a rate that is roughly proportional to the square of the operand size. Thus moving from a 32-bit representation to a 12-bit representation would result in a multiplier that is roughly 1/7th the area and power. Keep in mind that a 1-bit multiplier is simply an and-gate.

When verifying the bit accurate C, code coverage is essential. However it is important to note that traditional code coverage as used for software, such as the open source gcov or Coverity's coverage tool is not sufficient. The final hardware implementation may unroll loops, create multiple copies of execution units, and introduce pipelines. These operations will impact what the code coverage needs to be aware of. For example, if a loop is completely unrolled the loop control logic will not exist in the final circuit in the same way that it does in the algorithmic C code. The coverage analysis performed needs to account for this properly, or an incomplete verification will result.

Another optimization that can be performed is the pruning of the graph. Since the inferencing will be specific to a particular application, the CNN can be modified to eliminate calculations where the weights are 0. nVidia and others have found that this can increase the performance of the design and more than double its power efficiency [3].

F. RTL Implementation and Verification

The abstract algorithm can then be implemented in RTL. Our team did this using high-level synthesis (HLS). HLS has the benefit of enabling the exploration of various architectures, including different pipelining schemes, multiple parallel execution units, and memory access patterns [4]. Of course, this step could be performed manually as well, but one would find it more challenging to explore multiple design alternatives. Once in RTL, the TensorFlow graph in Python on the desktop can act as a testbench, both driving stimulus and checking for the correctness of results. This is done in a manner similar to how the C implementation was linked in to the TensorFlow network in an earlier stage. Once that data is received in the C code, it is passed to the logic simulator (or emulation system) through DPI calls. See Figure #3.

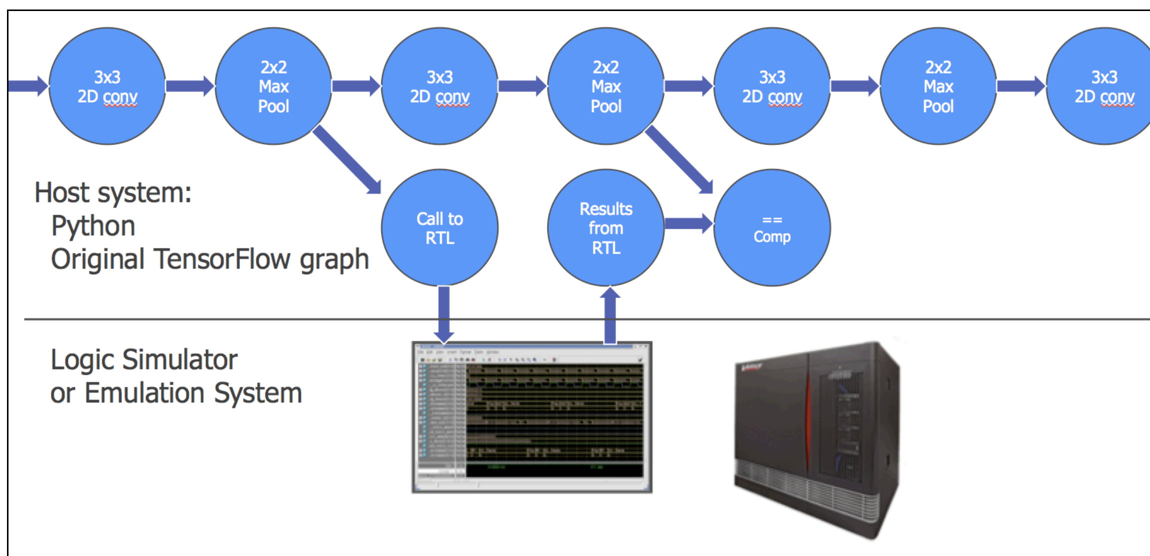

Figure #3 – RTL implementation running in parallel with original Python

Note that this does not provide any way to evaluate the overall performance or throughput of the final system, but does provide a way for determining the latency of an individual transaction through the accelerator. It is possible to get some idea on possible upper bound of performance. If that upper bound is not faster than the requirements for the system, then a re-evaluation of the architecture is warranted. For example, the Yolo-tiny algorithm requires 2.2 billion multiplies and 2 billion additions. If we create an

accelerator with 9 parallel multipliers and 8 adders, that produces one output value per clock when effectively pipelined, the fastest it can complete an inference is in 2.2 billion/9, or about 244 million clocks. Employing 4 of these in parallel would give us a lower bound of 61 million clocks per inference. With the RTL and the target technology library we can calculate the maximum clock frequency, and thus an approximation of the maximum possible performance.

For RTL generated using HLS, equivalence between algorithm and RTL can be done through formal verification – eliminating the need for much of the traditional dynamic RTL verification. The code that did the high level synthesis has performed a deep analysis of the bit accurate C code, and has intimate knowledge of the RTL produced. Using the information, sequential equivalency between the C code and the RTL can be mathematically proved. In some cases it will not be completely provable. In that case a partial proof can be used which will significantly reduce the need for dynamic simulation on the RTL. nVidia found substantial verification benefit by doing most of the dynamic verification in C and C++, and using formal techniques to prove equivalence between the high level algorithm and the RTL implementation. Overall, they were able to reduce RTL verification time by a factor of 500 [5]. For hand-generated RTL, applying formal methods will be more challenging. Without formal methods developers need to perform significant verification of the RTL.

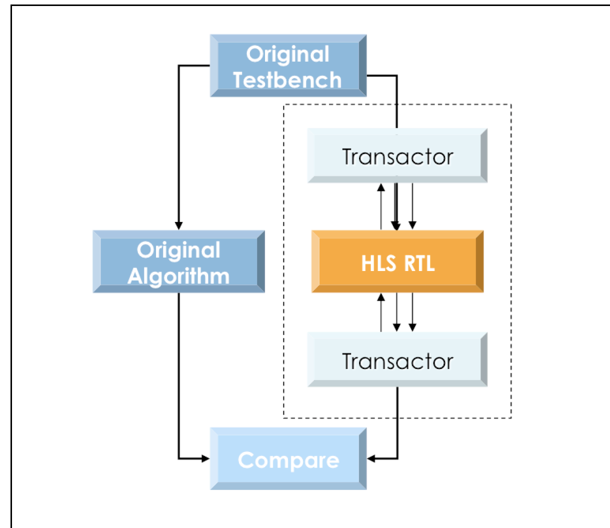

Figure #4 – verification environment that compares operation of the original algorithm with RTL

For those parts of the verification that were not covered by formal methods, we applied dynamic simulation of the RTL. Our HLS tool automated the collection of stimuli and responses from a C verification run and applied that stimulus and response checking to the RTL design, see figure #4. This made it easy to complete the verification. For developers building their RTL manually, a similar approach can be implemented.

The next step is to verify the platform as an embedded system. This means moving the TensorFlow graph and Python to a virtual machine (VM) that models the processor/memory subsystem to be used in the final implementation. There are various open source processor emulation systems (http://www.qemu.org and https://github.com/riscv/riscv-isa-sim) that can be used for this, although there is "some assembly required" to connect these to an RTL execution environment. Commercial offerings of environments that combine VMs with RTL execution systems are available from processor vendors (https://www.arm.com/products/development-tools/simulation/fast-models) and the major electronic design automation (EDA) vendors.

In this configuration the VM maps the physical address space where the accelerator is memory mapped. Any memory accesses to these addresses are not handled by the VM, but are passed to a C API. These

memory references are then passed through DPI to a bus interface transactor, which will drive activity into the RTL design.

It is also important to remember that the VM is not running in a clock cycle accurate fashion, therefore the timing of the data access will not match the final system. As a result this configuration cannot be used to verify final performance, throughput, response times, and other real-time aspects of the system (although many are tempted to draw such conclusions at this point). This stage of the verification primarily checks the embedded processor interfacing and software aspects of the system against the RTL of the accelerator. In this context the Python can still be used to identify discrepancies between the original algorithm (as executed on the target platform) and the accelerator. See figure #5.

This configuration brings in the cross-compiled application environment, bit accurate numeric representations, drivers, and final interfaces. Here the full software stack, Linux, Python, and supporting software, will all be running on the target processor, and correspondingly compiled for the target architecture. The final memory access patterns can be observed, resulting from the actual load and store instructions executing, and RTL accurate peripherals driving I/O cycles. Some VMs will include the ability to approximate the effects of caches, others will not. Memory caches will impact performance and data-flows.

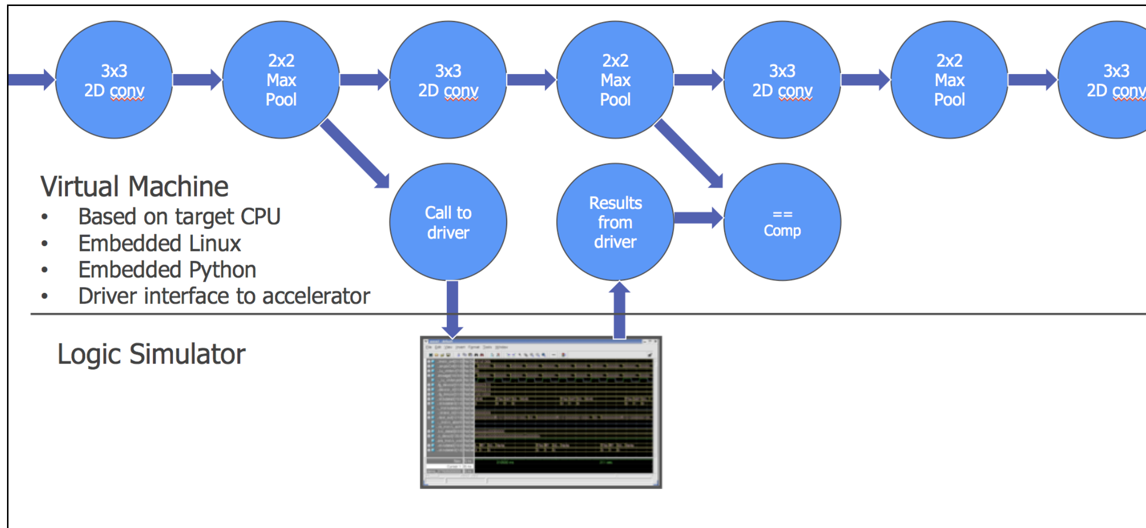

Figure #5 – executing TensorFlow graph on target VM with RTL accelerator running in parallel

While system-level power cannot be determined at this point, the power consumption of just the accelerator can be estimated and some optimizations can be done. Using the target silicon ASIC technology library and a preliminary layout, along with the stimulus from actual inferencing operations one of the commercially available power analysis tools can provide an approximation of power consumption for the accelerator as a stand-alone functional block.

At this stage there is an accurate view of the data movement through the system, and different approaches to minimizing the data transactions can be explored. Both in terms of performance and power, accessing data in off-chip DRAM is the most expensive operation that can take place. Stanford University's Mark Horowitz found that a DRAM access consumes 2,500 times more power than an addition operation [6]. During this stage we observed that accesses to the coefficient data in external memory was performed in a predictable, but non-contiguous fashion. By simply re-ordering the data in memory, four coefficients could be read in a single bus cycle, instead of one, affecting both performance and power consumption.

G. *Complete RTL System*

The final step is to run the complete system at the register transfer level (RTL). In order to get sufficient performance to run meaningful amounts of data through the design, emulation or FPGA prototyping systems must be employed. At this stage all performance related aspects of the accelerator and the surrounding system can be verified. It is also possible to evaluate the power consumption of both the accelerator and the overall system. In this configuration there will be an RTL representation of the processor, communication structures such as bus fabrics and interconnects, the accelerator, and any peripherals involved in the execution.

III. Conclusion

This paper has covered the verification of a machine-learning algorithm as it is deployed as an application specific accelerator. It starts with identification of the parts of the algorithm that are potential candidates for acceleration. Next, an algorithmic C implementation of the portion of the algorithm to be accelerated is created. This is verified in the context of the original implementation. This is followed by a bit accurate and implementation specific C implementation. Then an RTL implementation is derived from that, and it is verified against the original algorithm on the desktop, the same algorithm ported to the target, and finally against a full RTL system implementation.

IV. References

[1] M. Abadi, P. Barahm, J. Chen, Z. Chen, A Davis, J. Dean, et al. in Proceedings of the 12th annual Symposium on Operating Systems Design and Implementation (OSDI '16) Savannah, Georgia, U.S.A. November 2-4, 2016

[2] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You Only Look Once: Unified, Real-Time Object Detection, IEEE conference on Computer Vision and Pattern Recognition (CVPR `16) June 26th – July 1, 2016

[3] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S.W. Keckler, and W.J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017.

[4] J.P. Elliot, Understanding Behavioral Synthesis: A Practical Guide to High-Level Design, Kluwer Academic Publishers, 1999

[5] F. Sijsterman, J. C. Li, nVidia Closes Design Complexity Gap with High-Level Synthesis [online]. available https://www.mentor.com/hls-lp/resources/overview/nvidia-case-study-on-high-level-synthesis-hls--14def801-a98d-465e-9fdb-3d2b5cce1b25 [1 December 2018]

[6] M. Horowitz. Computing's Energy Problem (and what we can do about it). In International Solid-State Circuits Conference (ISSCC), pages 10–14, February 2014.