QWAC: Quantized Weight-Accumulating Combiner

Ethan Hicks[1]

[1]University of California

***Abstract*—As neural networks become more common, there is a need to accelerate computing power. QWAC is one such accelerator that quickly multiplies matrices against vectors. Based off Microsoft's Brainwave accelerator, QWAC utilizes quantization to decrease power and data movement while maintaining accuracy in neural networks.**

I. Introduction and Motivation

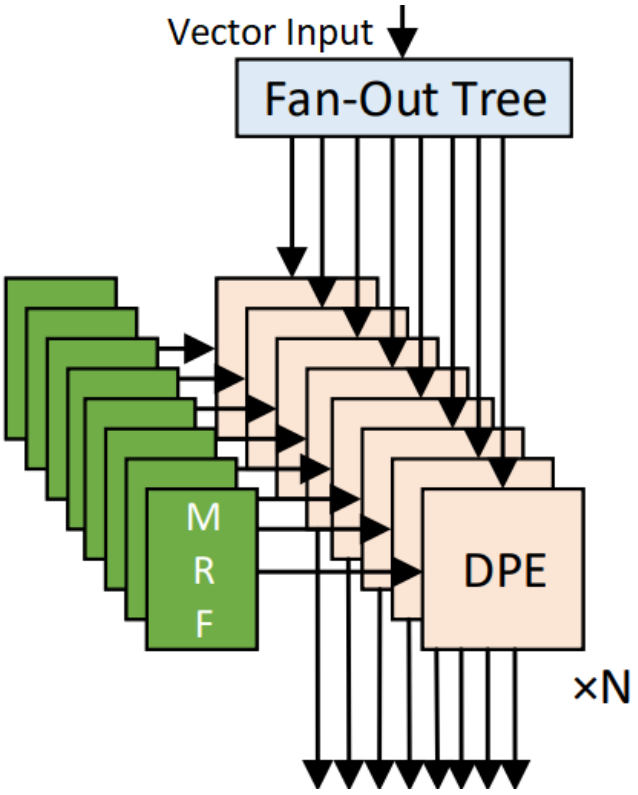
Driven by the increasing need to compute an increasing amount of data, Multiplication-and-Accumulate (MAC) ASICs and accelerators have emerged as a efficient method to compute, granting the engineers and teams total control over data movement, data density, computation speeds, etc. Many such acclerators and accelerator architectures have emerged, such as a the systolic array with Google's TPU[1] and Gemmini[2], or the adder tree with Nvidias NVDLA[3]. In Microsoft's Brainwave[4] a new kind of data movement is explored as the matrix is passed to each tile engine. This project is based off of this.

Microsoft's original implementation uses 16-bit floating point. In recent times, however, many neural network architectures have shifted towards using quantized values. Such a shift began with, and was proven by, Google[5]. The use of quantized values have emerged because the integer-only arithmetic reduces the power usage that floating point hardware requires while increasing speed as integer operations are faster than floating point for the same number of bits. Taking the idea of quantization and applying it to (a simplified version of) Brainwave leads to QWAC.

II. Changes between Brainwave and QWAC

A. API and Hardware Changes

QWAC implements some of the functions that Brainwave did. The purpose of this is to simplify the project while keeping a core of matrix-vector multiplication. Furthermore, the time and complexity that being at Microsoft provides for means that the engineering team can do more with the chip. For instance, they implement several non-linear functions into hardware itself. While these options certainly are nice to have, many are not used anymore and are thus out of date (for instance, the hyperbolic tangent function is put directly into the Brainwave accelerator). Maintaining the modernity of the current research, only the ReLU function is put directly into hardware. Not only is this cutting edge, but it is rather simple to implement in hardware. Seeing as vector-vector addition will take place regardless when computing a dot product (and thus makes sense to create a submodule for, thereby automatically creating the vv_add in the API), ReLU is thus created with a simple control signal.



Tile engine of QWAC. Note the input vector is copied to each dot product engine, while each matrix vector is unique to each DPE

Due to the quantized nature of the QWAC chip, there is no need to create an exponent storage network for the proper result, simplifying the architecture of the tile engine. Such a reduction of hardware further aids to reduce power draw when performing operations.

B. Memory Changes

Memory in QWAC is implemented naively. There is an integer number of matrices, an integer number of vectors, and an integer number of processed vectors, each with their own memory addresses. Driven by parametrization, the processed vectors and the matrices can be of a different bit-width than the input vectors, and the input vectors can be of a different length than the processed vectors. To make memory management easy in hardware, sending in an address will return the whole vector or matrix (the ease of memory management is also the reason why scalar values are not implemented in the API).

Hardware API					
Name	Description	IN	Out	In QWAC?	Note
v_read	Vector Read	Memory index	vector	Yes	-
v_write	Vector write	vector, memory index	-	Yes	-
m_read	Matrix Read	memory index	matrix	Yes	-
m_write	Matrix Write	matrix, memory index	-	Yes	-
mv_mul	Matrix vector Multiply	vector and matrix memory indexes	vector	Yes	-
vv_add	vector vector add	vector	vector	Yes	Hadamard
vv_a_sub_b	vector vector subtract	vector	vector	Yes	Hadamard
vv_b_sub_a	vector vector subtract	vector	vector	Yes	Hadamard
vv_max	Max element in a vector	vector	scalar	No	To
v_relu	Vector ReLu	vector	scalar	Yes	The only non-linear function due to ease of implementation
v_sigmoid	Vector sigmoid	vector	scalar	No	Complicated hardware; out of date as a non-linear function
v_tanh	Vector hyperbolic tangent	vector	scalar	No	Complicated hardware; out of date as a non-linear function
s_wr	scalar write	scalar, memory index	-	No	On chip memory isn't implemented for scalar adding
end_chain	end of instruction chain	-		No	Needed for integration in larger programming projects

TABLE I
COMPARING THE API BETWEEN THE TWO CHIPS

Furthermore, the integer-number of vectors and matrices exists to avoid a mapping problem in hardware, as each matrix and vector are of a predetermined, parameterizable size, QWAC memory management knows exacly how long a vector or matrix to read/write from a single address.

Microsoft's Brainwave sets itself up for dealing with memory mapping through Tile Engine array, and assuming that there will be many accumulations leading up to the final processed vector. Because everything is implemented in integer number of values, QWAC defaults to only one tile engine, that then returns the final processed vector afterwords.

While the exact specifications for the on-chip memory are not mentioned in the original paper, QWAC's on-chip memory is implemented through single clock synchronous read and write, allowing for both operations on the same cycle. The memory controller deals with potential hazards naively: always performing read then write. This ensures that the existing data is the one that is sent out upon request. Because the matrix in question stays within the register files of the tile engine, there is little potential for a matrix overlap until after the vector has been processed. Similar logic applies to the input activation vector: as it mostly remains unchanged until after operations have been performed upon it, there is little reason for it to be changed mid-op. Finally, the output vector is of a different type, of a potentially different length of the input activation vector, meaning that its reuse is rather unlikely. These three reasons demonstrate why a naive memory controller works.

III. CREATING AND BUILDING

The creation of this was not easy, and consisted of a number of unforseen steps that delayed progress and continue to create an overall lack-luster final outcome.

A. Makefile

Given a familiarity with Makefile driven development, it was assumed that there was a easy way to copy the makefile from a previous project and to mimic it. While this may be true for some, I could not figure out how to uses makefile to create a waveform via Synopsys. Past labs in the class have used 'make ...' and 'include=...', which is what I attempted to imitate. Such an imitation could not occur because it would require a different sourcing of /.bashrc, and a means and the knowledge to edit this file. Furthermore, as a specific example, the synopsys flag '-PP' – which seems to be to be the flag that tell Synopsys to create the waveform for debugging purposes that has been used in the past – has since been deprecated and indeed creates an error when used. While there must be a substitute for this flag, I genuinely could not find it and looking for it lead to many hours of no progress down a rabbit hole. To sum it up shortly, I could always use Synopsys to create a '.tbi' file, but I could never figure out how to take this file and turn it into '.vpd' file.

While my frustrations and inability to create waveforms and compiled files with Synopsys, other Verilog compilers on the market, Verilator and Icarus Verilog, seem to not provide good feedback for when compiling the program fails due to syntax errors (or otherwise). Furthermore, each one of these other very robust programs would require searching for flags and command line variables that would take as much work as Synopsys while providing the little to no help for compile-time syntax errors.

B. Vivado and Quartus

After more hours that I would like to admit trying to work with Makefile, I downloaded Vivado and used that as the debugging and waveform generating platform. The built-in waveform and RTL visualizer aided with development and easily created a debugging platform that I could use to fix my designs and more forward with the project. Furthermore, as Vivado simulates all generation of modules in a tree hierarchy, I could watch as the chip expands out word from the dot product engine all the way up to the MatVec multiplier.

Quartus was incorporated because the test FPGA was from Altera. I would reccomend sticking with Vivado. While manually creating an overall block diagram of the chip grants more control in the creation process, Quartus' waveform generation is not as intuitive as Vivados. As such, towards the end of the project I was using both IDEs to write, test, compile, transfer, plan, and compile again the code to the final FPGA – in this case a Cyclone V.

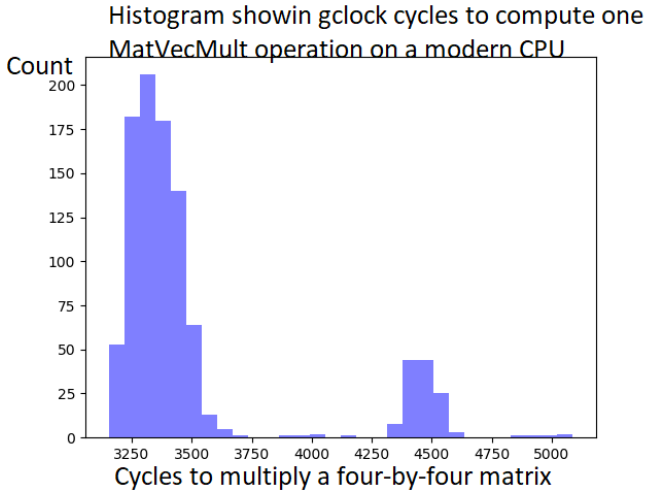
C. Naive Implementation

There are amazing abilities to both IDEs that I simply never touched, and once such example of this is the timing constraints file. As this ended up being more of a 'proof-of-concept' build rather than the whole accelerator like I wanted coming into this project, a timing constraints file never came into existence. As it stands, there is probably hold and setup issues and timing violations with the design.

IV. Testing

A matrix multiplier accelerator needs to be much faster than a computer doing the same task to justify the creation of it. Does this compute faster than a computer? Yes - about 10 times faster. Furthermore, the worst case scenarios on QWAC are far better than the worst case scenarios on a CPU, but this is undoubtedly due to the operating system overhead that takes priority over any testing code I wrote myself.

A. CPU Testing



Simple four by four MatVec multiplication clock cycle counting

Data on how many clock cycles are required to multiply a four by four matrix was gathered on a python script, using a hardware clock counting function written in C. The matrix was a numpy array, with the numpy ndarray operator '@' used for MatVec multiplication. Numpy, being the premier scientific computing package and being highly optimized, made a good choice for a compromise between ease of script creating and data collection. While there are two significant bumps around 3300 and 4400, most of the time it takes about 3300 clock cycles to compute this matrix multiplication. This script tested MatVec multiplication 1000 times.

B. QWAC testing

In order to test properly how fast QWAC computes would require the chip to be in much more of a working state than it is currently in, but testing still needs to commence to get an accurate idea of how well the basic functionality is.

The matrices are 8x4, hard-coded into the build of the FPGA. This is required because there is no communication between the host computer and the programmed FPGA. The vectors are also hard-coded in for the exact same reason. The vectors are 2x4, and are transposed in hardware. Since there is no communication once programmed, the on board switches are used to choose the matrix and vector combinations.

There is a simple implementation of a hardware clock counter that stops counting when a 'done' bit is passed into it. The done bit is toggled when the first element of the processed vector has been computed. This done bit is passed to the onboard LED array. Furthermore, the clock counter is 10-bit integer, chosen after some testing as a good compromise to be able to see the highest bits required – the LED array is only 8 elements long, and after choosing a 32-bit, a 24-bit, and a 16-bit counter, only the first 10-bits were needed.

The LED array in Little Endian format, here we can see that

the clock cycles to compute are {done_bit, counter[9:3] = 248 clock cycles

Overall, the chip is able to compute MatVec multiplication at a faster rate than the computer processor.

V. Conclusions and Future Work

While the final result is not as complete as I would like, there is still a hardware implementation of a Matvec multiplier that can be used to quickly and efficiently compute quantized vectors. Since this still strikes at the core of the project, I am happy with the work, but there is more work to be done.

First, there needs to be a way to connect the chip to the memory. This is vital for moving away from limited and hard coded values. The memory could be improved, but it is in a working state for now.

Next, there needs to be a way to choose what operation the chip is doing. That is, there needs to be a way to use the API rather than just having each API function implemented in hardware without anything using it.

Furthermore, there needs to be code written to communicate between the computer and the chip. This particular chip uses the FT 232H UART, and thus would need communication to be written in the FT D2XX driver platform. This would be the final step required to have a fully functioning chip

A. Special Thanks

Thanks to Sophia Shao, Hasan Genc, and Alon Amid for putting together a great course all semester long.

References

[1] J. et al, "In-datacenter performance analysis of a tensor processing unit," *44th International Symposium on Computer Architecture*, p. 17, 2017.

[2] G. et al, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," *University of California, Berkeley*, p. 15, 2019.

[3] Nvidia, "Nvidia deep learning accelerator," *Hot Chips*, p. 18, 2018.

[4] F. et al, "A configurable cloud-scale dnn processor for real-time ai," *ISCA*, p. 14, 2018.

[5] B. C. M. Z. M. T. A. H. H. A. D. K. Benoit Jacob, Skirmantas Kligys, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *Google*, p. 14, 2017.