Bitwise Convolution

Brian McMillin

Abstract

A series of observations concerning convolution with respect to image processing are presented.

The intention is to provide background for the development of more appropriate hardware and software for Convolutional Neural Networks. This includes reductions in complexity, memory requirements, computational requirements and processing time and power consumption.

TBD

Table of Contents

Convolution

The purpose of convolution is to transform an input stream (perhaps an image) into an output stream, with modifications based on a set of filter parameters. In particular, these filter parameters are applied based on certain aspects of (small) regions of the input. Applying the filter universally across the entire input is what generates the output.

Traditionally, we tend to think of input images as presented as a row, column sequence of pixels. A filter might be applied to a 3x3 subset of pixels, and generate a single output pixel. The filter would be conceptually moved across the entire input image to generate an output image.

A grayscale input image might be represented by a sequence of bytes representing per-pixel intensities. A filter for such an input might consist of a 3x3 set of scale factors. The scale factors would be multiplied by the appropriate input pixel values; the sum of these products would be the desired output pixel value.

This type of convolution works well enough to produce some useful (or at least pleasing) photographic effects such as Sharpen and Blur.

Filter Shapes

In general a filter (or more precisely a filter kernel) consists of a square matrix of scale factors.

The range of these scale factors is chosen based on matrix size, range of input values, and desired range of output values. Specifically, given a quantity of input values, each with a given range, we want to ensure that the sum-of-products output does not overflow.

The relative values of the elements within the filter conveys the desired filter properties, for example edge detection.

The use of a 3x3 filter is a common choice which creates a result by accumulating the sum of nine separate multiply operations. Even in the simplistic case of 8-bit grayscale image inputs, it becomes necessary to use some type of floating point representation to perform the required multiply-accumulate operation without loss of precision. Remember that the scale factors in the filter will be signed values thus allowing a particular product to make either a positive or negative contribution to the final sum. Maintaining the desired dynamic range in the output becomes even more problematic if the desired filter properties call for a sparse matrix (i.e., one with lots of zero elements).

There are alternatives to the 3x3 matrix that can be both more computationally efficient and more effective in accomplishing the desired task.

Consider: the entire purpose of a convolutional filter is to detect regions of interest and produce an output that either enhances or eliminates those interesting features. We would prefer to do as little computation as possible to reach this goal.

The traditional 3x3 filter uses all nine input pixels within its region to produce the resulting output pixel. Nine is not a nice number if you are used to trying to make efficient utilization of general-purpose hardware. So. Are all nine of those input pixels actually *necessary* to produce acceptable results? Well. The filter is intended to detect "features" in the input. "features" are different from their surroundings. The center pixel in the 3x3 matrix is the single pixel most likely to be highly correlated with the surrounding pixels. Therefore, it is least likely to make a meaningful contribution to the output. By simply ignoring it we can reduce our number of multiplies by 12.5%. Even better, we make the number of filter elements eight instead of nine, thus easing our power-of-two anxiety.

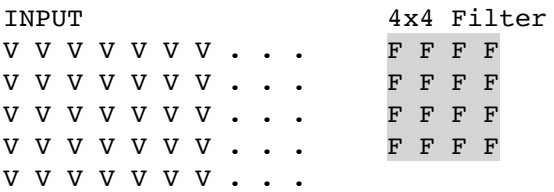
What else can we do? Suppose we call this option "8 out of 3x3". How about going further and trying for a "4 out of 3x3"? Two examples of possible versions are shown in the diagram. The utility of such things can only be determined via real-world examples. I leave this to the interested reader. The specific question to be answered is: Does the 56% reduction in computational requirements make up for the loss of precision in the result?

Some applications attempt to use 5x5 filters.

The diagram also includes a pair of "8 out of 5x5" possibilities. The potential advantage here is that a small number of computations could generate a result based on a sampling of input pixels from a larger area. This might, in some instances, yield similar results while using 68% fewer multiplies.

There is a tendency to want to use only filters that have odd dimensions. The desire is to match the output pixel with the "center" pixel of the input, and to have symmetric inputs surrounding that location. This is not a requirement, however.

We could just as easily use a 4x4 input area, for example.



As long as we are consistent in choosing an output pixel location relative to the input area we have satisfied the actual requirements of the convolution.

This also has the advantage of satisfying the power-of-two dimension requirement for efficient hardware utilization.

Concerning Image Boundaries

Usually it is deemed desirable for the output of a filter operation to be of the same size as the input. The definition of the convolutional filter would require taking inputs from outside the row, column boundaries of the input in order to accomplish this.

A crude solution to this problem would be to provide padding (possibly with zeroes) surrounding the input image. This arbitrary choice can cause artifacts at the edges of the output. A better solution would be to pad with duplicates of the edge pixels. This would be an improvement based on the assumption that the pixels just outside the input would be expected to be highly correlated with the pixels just inside the boundary.

Obvious artifacts on the edges of a Photoshop image would generally be unacceptable. Artifacts in the edge instances of convolutions used in neural networks are (most likely) irrelevant. The training of the network should cause them to be effectively invisible. Specifically, the degradation should be well below the noise level introduced by other aspects of the processing. One should expect that the Neural Network to be fully able to extract meaningful signal from the input in spite of these minor artifacts. And slowing the processing down to try to patch these infrequent instances is likely completely counter-productive.

Stochastic Processing

In several places I have pointed out the fact that real-world images are highly correlated from pixel to pixel. This statement applies not only to the input image, but to the convolved output image as well.

This high correlation in the output means that it might not be necessary to actually compute every pixel in the output. Specifically, a stochastic mechanism could be used to (on a random basis) simply duplicate the previous output pixel instead of computing the new value from scratch.

One could thus reduce the computational workload with a slight reduction in the accuracy of the result. The important thing here is that the computation requirements could be reduced much faster than the reduction in result quality.

Given a random number generator, the probability of the Compute vs. Duplicate result would establish the desired reduction in computational effort.

Reductio ad Absurdum

As we have seen, the range of input pixel values, the number of filter elements and the desired dynamic range of the output must all be carefully chosen. The goal is to retain the maximum amount of data, without truncation or clipping, while using the minimum resources in terms of both memory and computation.

The choice of each of these factors is non-trivial. There is a tendency to make the hardware implementations as general-purpose as possible because no one is certain what the actual requirements might be for a particular future application.

Since we have no idea what the actual numerical requirements are for filter parameters and Multiply-Accumulator precision, let us instead look at the conceptual minimum requirements.

What is the least amount of precision that we can get by with and still achieve useful results?

Exploration of the required precision has taken many forms. Typical is the use of 32-bit floating point weighting factors, often with the sum-of-products accumulated into a 64-bit value (to try to reduce roundoff errors) before producing a 32-bit floating-point result. Recognizing the extravagance of these GPU-based implementations, consideration has been given to the use of scaled 32-bit integers or even 16-bit values. Clever organization of 8-bit values into a floating-point representation has been tried.

Each of these approaches has been documented by their authors and the tradeoffs in result quality noted. Unfortunately, accurate understanding of these research results is hampered by the testing criteria. Real-world applications are so widely divergent that it is essentially impossible to understand the effect of a particular quality metric in a particular target application. The only way to achieve an accurate analysis is to fully implement and debug a particular approach using real data from the target environment.

Of particular interest to us at this point is the observation that each filter element causes the corresponding input pixel to increase or decrease the accumulated sum. This means that, in the limit, we could consider all elements of the filter kernel as taking on values of either -1 or +1. This is the effect that we could get from a truly binary filter using only one bit elements.

This is the approach used by XNOR-net in their Binary Convolutional Neural Networks.

Further simplification of the convolutional process is achieved by making the input image into a series of 1-bit pixels. This allows the input-pixel and filter-element to be operated on using using pure binary operations such as AND and XOR.

The use of binary operations eliminates the need for floating point computations and GPU implementations. The use of binary data reduces the memory requirements for the the system, the addressing complexity and the associated subsystem bandwidth.

Clever use of Binary Convolution allows the designer to implement extremely high speed inference operations. The speed and simplicity of the approach can form the basis of layer interconnections in Deep Neural Networks. The feasibility of a greater number of operations and lower per-filter memory usage means that deeper networks can be envisioned for a given model size (memory usage).

The binary filters can be viewed as "self-compressed" and not require the exotic data compression techniques used to make floating-point models fit into a manageable space.

Input Preparation

Since we are ultimately interested in the application of these convolutional techniques to Neural Networks, we are not concerned with the preparation of the "analog" input image data. We leave that initial step for future research.

A few hints might include:

- Neural networks are REALLY good at handling multiple interconnected layers, therefore the analog input need not be converted to just one binary input image.
- Use some image-wide or image-sequence (temporal) information to establish analog thresholds for converting individual pixels to single bit values prior to creating a binary input channel.

Memory Addressing

One of the biggest problems with the implementation of convolutional filters on "ordinary" (meaning von Neumann style) computers is the technique used for memory addressing. Memory addresses are treated as numbers and arithmetic operations (using the CPU) are performed to calculate offsets within data structures.

The input image is stored as a sequence of rows of consecutive pixels. Each of these pixels may have multiple components (red, green, blue, for example). Each of these components may need to be fed to a different kernel within the convolutional filter. In order to perform the convolution operation, each of these elements must be located (through computations carried out by the processor) and the resulting address passed to the memory subsystem.

The memory subsystem must be able to retrieve the desired data. The sequence of addresses presented form an essentially random pattern as far as the memory system is concerned. Sometimes there are near-sequential accesses, sometimes there are widely separated accesses, and occasionally there are accesses to recently used locations. This creates many opportunities

for a hardware designer to develop creative caching architectures to add a fast layer to otherwise slow main-memory accesses.

Key to getting speed out of the memory subsystem would be the ability of the software algorithm to issue pre-fetch instructions, without processing penalty, to load the cache subsystem in anticipation of the essentially random reads that will be needed shortly. This will allow the caches to be smaller that they typically are for general-purpose use now. Recognizing that as the filter "moves" across the image, certain pixels will not be needed again and others will still be used as part of the next scan line is something that the software algorithm is well aware of but that would be essentially impossible to automate in hardware.

The memory access pattern of a typical convolution algorithm represents an almost worst case situation as far as the memory system performance is concerned.

Key failings with the typical architecture are that

1. More time is spent computing addresses than performing the algorithm
2. Memory accesses drag along bunches of unused data adjacent to the desired item
3. Hardware that pre-fetches or retains anticipated data usually guesses wrong

Serialization

Convolution is a fundamentally parallel concept. The description of "moving a filter across an image" is a simplification to allow easy description of the underlying process. In reality every output pixel could be released as soon as the corresponding handful of input pixels became available.

In the case of real-world implementations we are often constrained by the inherent slowness of a camera interface. Data becomes available for processing at (for example) 30 frames per second. Applying a filter to this stream should be performed continuously, using very simple processing hardware. The desired output frame should be streamed at the same bit rate as the input, only with a delay of a few scan lines at most.

This is quite different from the traditional "filter the image" approach in which an entire image is loaded into memory and the the fastest available processor (maybe a GPU) is unleashed. The processor speed is critical here because many filters may need to be run and all of the processing time is adding latency to the results.

Recognizing the conceptual parallelism and implementing it in the lowest levels of a serial stream process allows us to reduce the hardware requirements and level out the CPU load, memory access requirements and power consumption.

Strides

For object detection and recognition tasks it is desirable that a particular filter be able to detect a target pattern at a range of scales within the image. Simple filters as described so far would be incapable of this - One filter for each size pattern would be required.

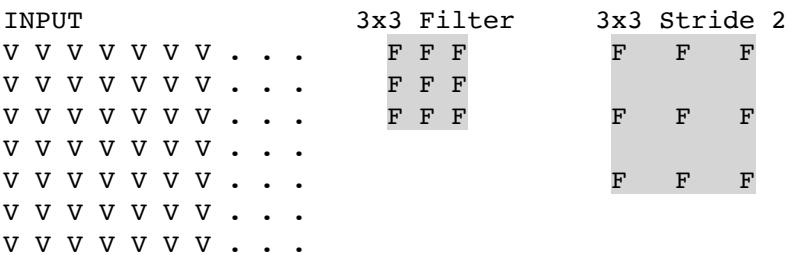
It is possible to approximate the desired scale independence by rescaling the input image. I.e. change the input size until it matches that required by the detector.

A simple way to approximate the required scaling is to use strides as the input data is fed into the system. The filters described so far can be considered to have a stride length of 1. That is the pixels fed into the filter are all directly adjacent to each other and there is a 1-to-1 relationship between the dimensions of the input and out images..

A stride length of 2 could describe a system in which only the output from every-other filter filter across a row and every-other row would be used to compose the output. This provides a reduction in the number of calculations and the size of the output by a factor of four.

The next layer would have less data to manipulate and would use filters trained on this scaled-down data.

An alternative implementation of the stride concept would use more widely spaced input pixels to the filter, but run the same number of operations and generate the same number of output pixels as before.



The filter slides row-by-row, one pixel at a time as before but the computations are performed on more widely spaced input pixels. This would achieve an effect similar to processing with a sparse 5x5 filter. The potential benefit of such an arrangement is that the memory access hardware would be able to more efficiently select the input data required for each filter computation. And the *a priori* knowledge of the locations of the zeroes in the filter would reduce the size of the filter array itself.

These two concepts involving input and output strides could certainly be combined in a particular implementation. The stride [input, output] tuple would be part of the filter description. Hence, one would refer to filter shape as "**`3x3 stride [1,1]`**", for example.

Ultimately, the same amount of input data would be presented to the system in all cases. Efficiency is gained through optimization of the redundant data movements and computations required as the output is produced.

It is expected that many different output images would be produced "simultaneously" with a single pass through the input image. This parallel application of multiple filters would minimize both memory requirements and computational latency. The resulting multiple outputs would be presented to the next layer of the deep network.

Obviously, these concepts would apply to the convolutions used in all the layers and layer interconnects of a deep net, not just the ostensible “input” layer that we have been referring to.

Our present interest pertains to algorithms that implement convolutions in binary neural networks. This amounts to organizing the bits appropriately and expeditiously for processing with simple bitwise operations.

One of the limitations of this pure-binary environment is that the filter elements must exist in only two possible states: we have chosen {-1, +1}. This, unfortunately, means that there is no possibility of a pure-zero filter element. This is not necessarily harmful - it does require a certain amount of revision to our typical thought processes. Specifically, it means that the filter elements can never form a sparse matrix.

A certain amount of this limitation can be addressed using the stride concept. A given filter can cover a larger input area and yet retain the smaller storage and processing requirements offered by a sparse matrix. And we retain the advantages of the SIMD parallelism described herein.

Given a proper training environment, one expects that even more complicated combinations of input / filter / output organizations would be possible. For example, the well-known filter-compression technique of using a bit-vector to indicate the location of zeroes in the matrix, followed by the (small) number of actual non-zero element coefficients. We expect that different filter shapes, different stride parameters, different patterns of zero-elements and different options for controlling stochastic computation are all learnable parameters for an advanced Machine Learning environment.

Non-Integer Strides

As described, an integer usually describes the interval between elements. This is generally convenient but represents a significant limitation in the ability to control

- The relative size of the output frame compared to the input frame
- The scale of the filter relative to the input frame
- The amount of overlap between successive applications of the filter to the input frame

From a hardware standpoint one could envision simple strides being implemented in the form of a shift register for the incoming data and a counter to select every Nth element.

It would be possible to enhance the versatility of this stride-generation mechanism by implementing an additional second counter with a different period. The interaction of these two counter periods would produce a syncopation.

The resulting complex period between selected input elements would allow for a greater choice for all three of the concerns listed above.

Data Structures

As mentioned before at some length, addressing data for convolution algorithms is problematic. The essentially random access pattern poses a challenge for most hardware.

It is incumbent on the algorithm designer to create data structures that minimize the access challenges to the underlying hardware. The use of single-bit operations is a tremendous boon to algorithm design. Given hardware that supports (for example) 64-bit integer representations and associated binary operations we have the opportunity to perform 64 binary operations, in parallel, during a single instruction cycle. This is almost painless Single Instruction Multiple Data (SIMD). All we have to do is arrange to get the data into the correct initial configuration. And not shoot ourselves in the foot by having to resort to some kind of operation on single bits or bit groups someplace in the algorithm.

The traditional, CS-101, concept of memory addressing is deeply ingrained. "Everyone" knows that a memory address accesses a word. You perform arithmetic on words and save the results at addresses. Unfortunately, this deep conviction cannot help us when we are trying to deal with bitwise operations over pixel arrays.

We can perform massively parallel bitwise operations on 64-bit words, but if we try to use a hardware arithmetic operation such as ADD we will destroy our carefully groomed parallel data. What we need is the necessary arithmetic operations that operate on a set of registers but retain the bit-at-a-time separation within the registers.

Consider. We have two single-bit binary values **A** and **B**. We wish to add them and get two result bits which we will call **Sum** and **Carry**. Grade-school stuff.

```
A          0 . . .
B          1 . . .
===========
Carry      0 . . .   This is A AND B
Sum        1 . . .   This is A XOR B
```

Now suppose we have 8 pairs of single-bit binary values. We want to get the sums of each pair.

```
A          0 0 1 1   1 0 1 0
B          1 0 1 0   1 1 0 0
===========
Carry      0 0 1 0   1 0 0 0   This is A AND B
Sum        1 0 0 1   0 1 1 0   This is A XOR B
```

We have computed eight single-bit binary addition operations using two machine operations, neither of which was an add. Obviously, given hardware with 64-bit integer registers, we can scale this up and achieve effective speeds found in massively parallel architectures.

Furthermore, these simple binary operations should be the fastest available instructions on a given set of hardware - because there is no carry propagation such as would be found in an

integer ADD instruction. Whether ADD instructions *actually* become slower due to carry propagation depends on individual processor clock utilization. But it makes the point that the hardware ADD instruction and all of its carry-propagation hardware is not strictly necessary in the first place.

Bit Counting (Horizontal)

Now, consider that at some point in our binary convolution algorithm we have a set of bits and we wish to count the number of 1's. A naive approach might be to use a program with shift, conditionals and add to create the sum.

Input Bits	1	0	1	1		0	1	0	1	**Output Sum** 0 1 0 1

An improved algorithm might use a sequence of shift, mask and add operations in a fast binary-tree style approach, like this:

Input Bits	1	0	1	1		0	1	0	1	
Mask Odd Bits	0	0	0	1		0	1	0	1	
Shift and Mask	0	1	0	1		0	0	0	0	(1 bitwise shift right)
Add	0	1	1	0		0	1	0	1	Sum of bits in each pair
Mask Odd Pairs	0	0	1	0		0	0	0	1	
Shift and Mask	0	0	0	1		0	0	0	1	(2 bitwise shifts right)
Add	0	0	1	1		0	0	1	0	Sum of bits in each quad
Mask Odd Quads	0	0	0	0		0	0	1	0	
Shift and Mask	0	0	0	0		0	0	1	1	(4 bitwise shifts right)
Add	0	0	0	0		0	1	0	1	Sum of bits in each byte

Thus, the 8-bit example illustrated here (giving a 4-bit sum) could be implemented as 8 parallel computations using the same number of instructions. Of course, this still uses the processor's 64-bit integer ADD instruction, but we are assured that no carry within the **Sum** will propagate past 4 bits. This may allow some increased performance, depending on the hardware.

This example presumes that we have somehow gotten all of the required input bits packed into a single word and that we want the resulting sum(s) to be in a traditional power-of-two field grouping within a result word.

Actual instruction usage for doing this 8-bit counting operation would be:

(6) AND operations for the required bit masking
(7) SHIFT operations for bitwise positioning
(3) ADD operations to compute the sum with hardware carry as needed

Thus, for 64-bit hardware with this type of SIMD operation we can achieve the required sum-of bits in an average of two machine cycles per result.

Perhaps we can use the insights of the previous bitwise separation concept, including the elimination of the hardware ADD instruction, to achieve even greater effective speeds.

Bit Counting (Vertical)

Suppose that we configured the inputs and outputs into a set of 12 registers like this:

```
A     1 . . .   input bits in separate registers
B     0 . . .
C     1 . . .
D     1 . . .
E     0 . . .
F     1 . . .
G     0 . . .
H     1 . . .

s8    0 . . .   result bits in separate registers
s4    1 . . .
s2    0 . . .
s1    1 . . .
```

The tricks to this algorithm are:

1. We use the previous idea of XOR to sum two binary values and AND to get the carry.
2. We recognize that processing the input bits **A** to **H** is an increment - not really an add.
3. We know *a priori* how far the carry could propagate, so we trim unnecessary steps.
4. We explicitly set the value of each **s?** the first time we use it - no extra initialization.

Incrementing a 4-bit result in this configuration by brute force 7 times might require as many as 49 operations. By suppressing the unnecessary carry-propagating operations we can reduce this to a total of 30 operations.

Using this technique with 64-bit register operations we can compute the sum of 64 8-bit values in 30 clock cycles. This is effectively 2.1 summations per clock cycle - more than a 4x improvement over the previous algorithm.

The algorithm presented here should clarify the required sequence of machine instructions. I have named 3 temporary registers **c1**, **c2** and **c4** to hold carry-out values. Real implementations should be able to reduce the register requirements by folding the temporaries into other unused registers.

The columns are the destination registers. The sequence of operations, read left to right and top to bottom will compute the desired results.

c1	s1	c2	s2	c4	s4	s8
	A XOR **B**		**A** AND **B**			
s1 AND **C**	**s1** XOR **C**		**s2** XOR **c1**			
s1 AND **D**	**s1** XOR **D**	**s2** AND **c1**	**s2** XOR **c1**		**c2**	
s1 AND **E**	**s1** XOR **E**	**s2** AND **c1**	**s2** XOR **c1**		**s4** XOR **c2**	
s1 AND **F**	**s1** XOR **F**	**s2** AND **c1**	**s2** XOR **c1**		**s4** XOR **c2**	
s1 AND **G**	**s1** XOR **G**	**s2** AND **c1**	**s2** XOR **c1**		**s4** XOR **c2**	
s1 AND **H**	**s1** XOR **H**	**s2** AND **c1**	**s2** XOR **c1**	**s4** AND **c2**	**s4** XOR **c2**	**c4**

As with all true SIMD architectures the operation is totally deterministic. There are no branches in the code and all "conditional" operations are accomplished strictly through the binary data. Thus, the program is a straight-through linear sequence of operations with no loops. This means that the hardware instruction accessing, decoding pipeline and caching should have the easiest possible time. Again, this brings some of the highly-touted features of GPU architecture to a much simpler CPU-based algorithm.

Increment (Vertical)

Let us think about a binary increment operation. We would like it to be conditional, in that we have an addend **A** which determines whether we increment or not.

```
Sum        0 . . .
A          1 . . .
===========
Carry      0 . . .   This is Sum AND A
Sum        1 . . .   This is Sum XOR B, our new total
```

Suppose that we have a 4-register total. Each successive subtrahend is the previous borrow.

```
s8    0 . . .   result bits in separate registers
s4    1 . . .
s2    1 . . .
s1    0 . . .
```

And we allocate temporary registers for carry. For clarity they are **c1**, **c2** and **c4**. Only two temporaries are ever needed at a time so **c1** and **c4** could be the same physical location.

c1	s1	c2	s2	c4	s4	s8
s1 AND **A**	**s1** XOR **A**	**s2** AND **c1**	**s2** XOR **c1**	**s4** AND **c2**	**s4** XOR **c2**	**c4**

Decrement (Vertical)

We have created a conditional-binary-increment algorithm. Now consider a similar technique for conditional-binary-decrement.

We have two single-bit binary values **Total** and subtrahend **S**. We wish to subtract (**Total** - **S**) and get two result bits, the new **Total** and **Borrow**.

```
Total     0 . . .
S         1 . . .
==========
Borrow    1 . . .   This is S AND (NOT Total)
Total     1 . . .   This is Total XOR S, our new total
```

Suppose that we have a 4-register total. Each successive subtrahend is the previous borrow.

```
t8   0 . . .   result bits in separate registers
t4   1 . . .
t2   1 . . .
t1   0 . . .
```

And we allocate temporary registers for borrow. For clarity they are **b1**, **b2** and **b4**. Again, only two temporaries are ever needed at a time so **b1** and **b4** could be the same physical location.

b1	t1	b2	t2	b4	t4	t8
S AND (NOT **t1**)	**t1** XOR **S**	**b1** AND (NOT **t2**)	**t2** XOR **b1**	**b2** AND (NOT **t4**)	**t4** XOR **b2**	**b4** AND (NOT **t8**)

Decrementing is not as clean as incrementing for two reasons.

- We cannot short-circuit any of the operations since even the very first decrement propagates to every bit in the total.
- The generation of the borrow requires the negated version of the previous value, hence the NOT. This is problematic on typical hardware.

Our increment requires 6 machine instructions, the decrement needs 11.

Two-Input Gates

Looking at binary logic operations from the standpoint of programming languages and physical hardware it is common to have logic operations performed on two inputs. There are sixteen possible such operations, summarized in the following truth tables.

ZERO	AND	?	B	NIMP	A	XOR	OR	NOR	XNOR	~A	IMP	~B	?	NAND	ONE
0 0	0 0	0 0	0 0	0 1	0 1	0 1	0 1	1 0	1 0	1 0	1 0	1 1	1 1	1 1	1 1
0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1
✔	✔		✔		✔	✔	✔			✔		✔			✔

The operations marked with ✔ are commonly available as single machine instructions.

The **A** AND (NOT **B**) is, in fact, the NIMP or NOT Implication logic operation. No current general-purpose hardware implements the implication operations as single machine instructions.

It is likely that future hardware designed to accelerate the use of data structures such as we are discussing would do well to consider implementing some of these additional instructions.

Programming Language Limitations

Programming languages are expected to provide a generally compact and legible notation for describing algorithms and data structures. The notation should be easily written and understood by human beings. It should also convey precise instructions for the creation of machine code.

Unfortunately, none of these expectations are achieved in the present instance.

There is no existing notation that describes the data structure that I have just used. “Variables” whose value is an ordered sequence of individual bits selected from separate machine words do not exist in the literature.

The algorithm involved, even in the brute-force instance of incrementing such a “variable”, requires an understanding of number theory and must be expressed in assembly language.

The fact that the goal here is parallel (SIMD) operations on many of these “variables” at a time makes it even less likely that an existing programming notation would be appropriate for describing these operations. Or be understandable to anyone trying to decipher it.

Suppressing instructions whose results are never used is something that we would expect of a modern language compiler. Of course, programs written in assembly language are generally not subject to this type of optimization. Likewise for the expected register-usage optimization mentioned earlier.

Thus, we are left with the worst of all worlds. We must describe both our data structures and algorithms in the most obtuse way imaginable. And we can get no automated assistance in the memory allocation, register assignment or code optimization processes.

Working with these algorithms will therefore require exhaustive simulation, test cases and regression testing. The types of errors that human beings are likely to introduce during program development are invariably obscure and potentially impossible to isolate without careful attention from the earliest stages of design.

Members of a development team must be capable and confident when making changes to the software. And some mechanism must exist for catching the inevitable errors as quickly as possible.

Continued development and ongoing maintenance of software such as this requires clear and accessible documentation and training materials. All team members must be able to understand the philosophical ***why*** behind an algorithm, not just the opaque and impenetrable ***how*** embodied in the code.

Ruthless adherence to a design, testing and documentation methodology will be required.

Image Data Organization

Our goal is to use these techniques to effectively perform parallel convolutions on images presented as sequential rows of bits. Suppose we have a few rows from an image in machine words like this:

```
V      V V V V V V V . . .
W      W W W W W W W . . .      An example "8 out of 3x3" filter
X      X X X X X X X . . .      highlighted in an input image.
Y      Y Y Y Y Y Y Y . . .
Z      Z Z Z Z Z Z Z . . .
```

We would like to arrange to be able to make use of these values as directly as possible as input to the previous bitwise summation algorithm. We can achieve the desired configuration using only LOAD, and SHIFT instructions. Our goal for input registers is to look like:

```
A      W W W W W W W . . .      Versions of Row W
B      W W W W W W . . .
C      W W W W W . . .
D      X X X X X X X . . .      Versions of Row X
E      X X X X X . . .
F      Y Y Y Y Y Y Y . . .      Versions of Row Y
```

```
G       Y Y Y Y Y Y . . .
H       Y Y Y Y Y . . .
```

By storing slightly shifted version of our three rows of input data we have aligned the desired pixels into bitwise columns. The convolution operations (in our example: bit summing) then proceed in parallel.

Optimization (redux)

The designation of eight input registers **A** - **H** is only a conceptual convenience. Perusal of the bit-summing algorithm reveals that each "register" is used in a pair of instructions and then never accessed again. And they are accessed in order from **A** to **H**. Since (for example) **A**, **B** and **C** are just variously shifted versions of the image data that we refer to as **W**, it turns out that we do not really need all these registers. Likewise for all the other things referred to as Input.

It turns out that for a real-world implementation we need only a single "Input" register which is loaded from the image data in main memory and manipulated as needed within the summing algorithm.

Our 64 filter output sums will reside in four actual result registers.

Thus, we are able to fully implement a streaming version of this portion of the image convolution. Every input image row will be accessed three times in quick succession. Scanning from top to bottom, the rows will never be needed again.

Edge Effects (redux)

As noted in the earlier discussion of Image Boundaries, the boundary of the image will generate artifacts in the convolved output. Specifically, the first and last bit of each image row will not be technically correct.

In our example we are able to achieve full accuracy for 61 of the 64 filters. The two-bit-shift operations would be expected to rotate in pixels from the next word in each image row. If this additional bit rotation was included we would generate correct results for all but the first and last filter: 62 out of a possible 64.

As mentioned, it may turn out that the significance of artifacts such as these may be small enough that they can be safely ignored. The Neural Network may simply "learn its way around them", just as a person completely ignores the muntin bars between the panes in a window.

Binary Convolution

Given these out-of-the-box data structures and algorithms, we can now turn to the actual Binary Convolution.

Our specific goal is to produce a single-bit output pixel based on 9 input pixels and 9 filter coefficients. Our image pixels will be binary values taken from the set {0, 1}. Our filter

coefficients will be taken from the set {-1, +1} but represented by the binary values {0, 1}. We will compute the sum of 9 products; the result will be the sign of that sum.

The truth table for each of the products that we will compute is:

	Image Pixel 0	**Image Pixel 1**
Filter = 0 (meaning -1)	**0**	**-1 (decrement sum)**
Filter = 1 (meaning +1)	**0**	**+1 (increment sum)**

TBD

Deep Neural Networks

TBD

Inference

TBD

Training

TBD

Hardware Considerations

The bulk of this paper argues in favor of simplified data and algorithm requirements. The use of single-bit image and filter elements allows for the elimination of floating point operations Parallel operations on binary data can be accomplished by a subset of the hardware found on a typical CPU. Clever data organization and algorithm design minimizes the limitations posed by memory architecture and bandwidth. The elimination of GPU requirements reduces complexity, power and thermal issues.

A **Bad Idea**, replicated 1024 times and clocked
at 3GHz does not become a **Good Idea**.
It may solve the target problem, though.

TBD

Summary

This document has presented alternative algorithms for application to Binary Convolution using CPU hardware.

TBD