Symbol definitions

- N = the number of input features,
- M = the number of hidden neurons (in our case it is always 40),
- C = the number of output neurons/ number of classes
- S be the number of samples in the dataset,
- x_i be the ith input feature,
- x_i be the ith input feature,
- D^i be the i-th sample of the dataset,
- D^i_j be the value the j_th_ input feature takes on the i_th_ sample of the dataset,
- h_i be the i_th_ hidden neuron, also used to denote it's output value before binarization,
- s_i be the i_th_ hidden neuron's output after binarization, so $s_i = h_i \geq 0$,
- y_i be the i_th output neuron, also used to denote it's output value,
- $W1$ = the weight matrix of the first layer,
- $W2$ = the weight matrix of the second layer,

Rows represent neurons and columns represent input activations, so $W1_{i,j}$ is the weight of the first layer that corresponds to the connection between the input feature x_j and the neuron h_i.

Combinatorial fully connected implementations

Two initial approaches for implementing the fully connected BNNs in a fully combinatorial single-cycle datapath are compared. Only the first layer differs between them, the second layer stays unchanged.

Positive-Negative Sum

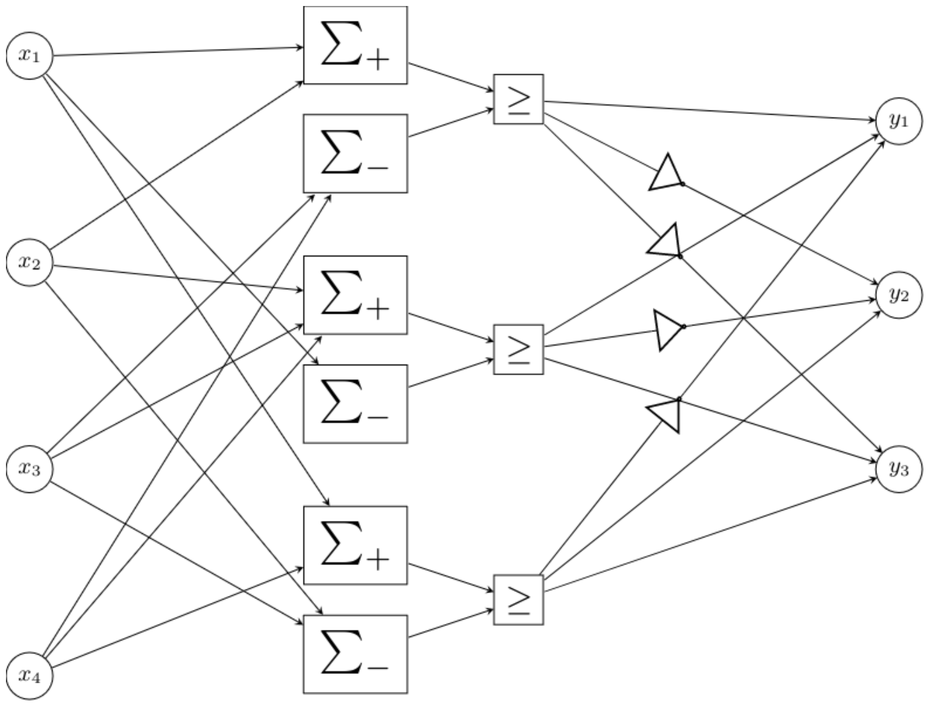


Figure 1: Implementation with positive and negative sums split

For each neuron in the first layer two sums are calculated. Σ_i^+ is the sum of the input features for which the connection with the i_th_ hidden neuron has a positive weight , whereas Σ_i^- the sum of those that have a negative weight associated. The two sums are then compared and if the positive sum is greater than or equal to the negative the output of the neuron is 1, otherwise 0.

$$\Sigma_i^+ = \sum_{j=0}^{N-1} x_j [W1_{i,j} > 0]$$

$$\Sigma_i^- = \sum_{j=0}^{N-1} x_j[W1_{i,j} < 0]$$

$$h_i = \Sigma_i^+ \geq \Sigma_i^-$$

Sample code snippet:

```
assign positives[0] = + feature_array[1] + feature_array[2] + ...
 ↪  + feature_array[10];
assign negatives[0] = + feature_array[0] + feature_array[3] +
 ↪  feature_array[5];
assign hidden[0] = positives[0] >= negatives[0];
```

The reasoning behind splitting the sums is that keeping the operations to only using unsigned positive numbers and only using addition and not subtraction means simpler operations are required, and that may lead to a smaller footprint.

For each neuron of the output layer it's value is calculated by summing the output of hidden neurons. The binary output of the hidden neuron $s_j$is added as-is to the sum of the output neuron y_i in the case that the weight of their connection $W2_{i,j}$ is positive and it's binary inverse is added to the sum if $W2_{i,j}$ is negative. This is equivalant to the sum of the xnor between the output vector of the hidden layer and the weight vector of the output neuron.

$$y_i = \sum_{j=0}^{M-1} \begin{cases} s_j, & \text{if } W2_{i,j} > 0 \\ \neg s_j, & \text{if } W2_{i,j} < 0 \end{cases}$$

Code sample:

```
assign scores[0*SUM_BITS+:SUM_BITS] = + hidden_n[0] + hidden[1] +
 ↪  hidden[2] + ... + hidden_n[39];
```

Signed sum

In this version a single sum is calculated for each neuron. If the connection between input feature x_j and hidden neuron h_i has weight $W1_{i,j} = 1$ it is added to the sum, otherwise it is subtracted from it. Basically adding the feature multiplied by either 1 or -1 is hard-coded as the resulting addition or subtraction correspondingly. The result is then compared to zero to give the binarized output of the neuron. Since the result is a signed number, this just means taking the sign bit.

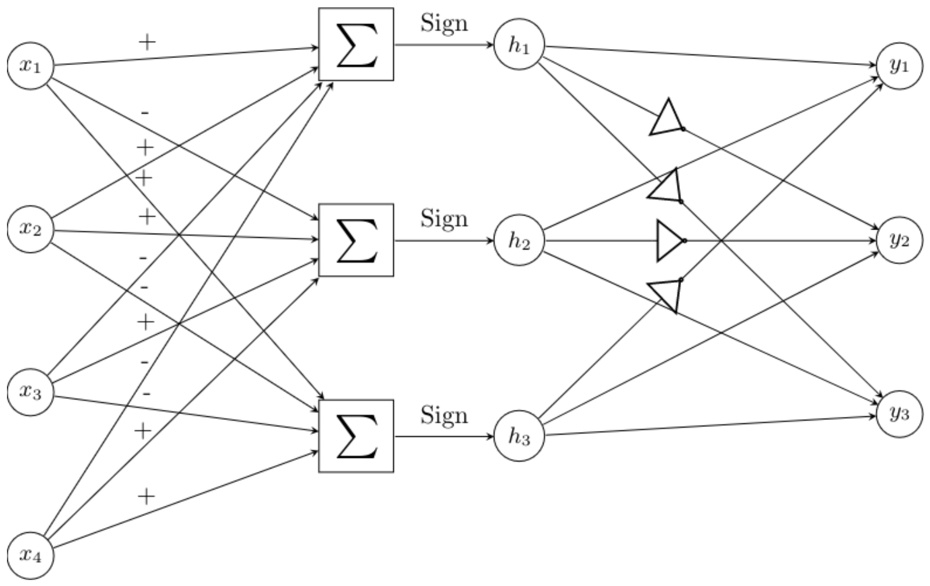


Figure 2: Implementation a single sum per neuron

$$h_i = \sum_{j=0}^{N-1} \begin{cases} +x_j, & \text{if } W1_{i,j} > 0 \\ -x_j, & \text{if } W1_{i,j} < 0 \end{cases}$$

Code sample:

```
wire signed [8:0] intra_0;
assign intra_0 = - feature_array[0] + feature_array[1] + ... +
 ↪  feature_array[10];
assign hidden[0] = intra_0 >= 0;
```

The second layer's implementation is not changed from how it is described above.

Discussion

My prior expectation was that splitting the features into two sums to avoid subtractions would lead to better results than keeping all the operations for a neuron to a single expression by the reasoning summarised previously. In fact it turns out that using a single sum has area and power requirements 20-30% lower than using two.

My explanation for this is that keeping terms in separate expressions prevents the compiler from effectively locating and reducing shared subexpressions during synthesis. For example, if a neuron's expression contains $+x_5+x_6-x_7$ and a different neuron contains $-x_5+x_6+x_7$ then the result of x_5-x_7 can be used for both neurons, but if x_5 and x_7 are not in the same expression, as they would not be in the positive-negative sum implementation, this arithmetic optimization is not used by the compiler. Additionally the two sums need comparators to produce the binarized output of the neuron, whereas in the case the signed sum this corresponds to the sign bit of the sum which means no additional hardware.

Minimum range bit-width reduction

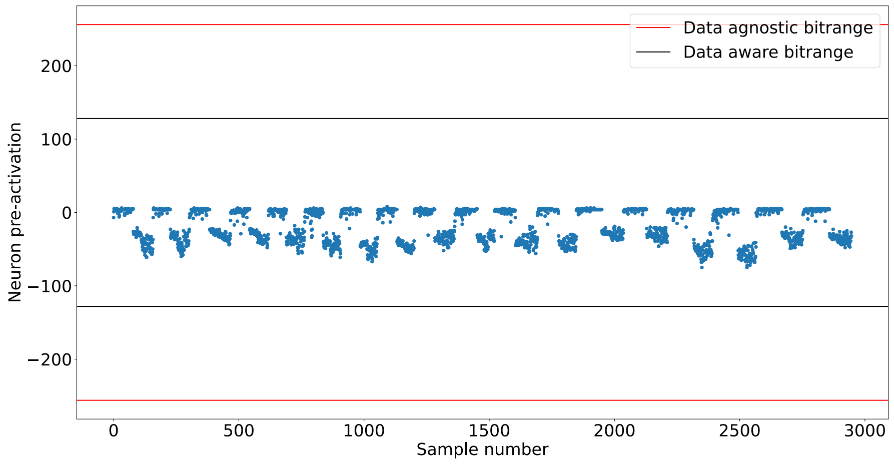


Figure 3: title here

The reasoning goes that if the bitwidth that is demanded for the total sum of the features is reduced, the bitwidth that partial sums it depends on require lower on the adder graph is also reduced. As a consequence, adders need to accomodate less bits and less logic ought to be implemented for them.

At least in theory, the minimal circuit to implement the calculations of the sums of the first layer with reduced widths for the results should be strictly equal or smaller than the one with full widths. This is because given a circuit that implements the full width calculations, the reduced width results can be taken by selecting the bitrange of that width from the least significant bits of the full width result. This is simple wire selection and does not require any additional hardware, therefore reducing neuron widths can never require additional logic.

So far in the first layer the bitwidth of the total sum h_i of the neuron is set to be large enough to fit any value that may come up as the result of M additions and subtractions of 4-bit unsigned numbers. The assumption is that this range is significantly wider than the range of values the neuron actually takes during evaluation of typical samples. This would mean the bitwidth can be lowered without errors due to overflows or underflows coming up in actual use of the design, and this lowering would improve performance.

To test this the total sum of each neuron h_i is calculated for each sample in the dataset. I get the minimum and maximum of these values.

Since all values for the total sum to take are contained in the range between those two, the arithmetic operations need not accomodate any range larger than that. Let H_j^i be the value of h_i when evaluating the j_th_ sample of the dataset, and wh_i be the bitwidth of the i_th_ hidden neuron.

$$h_{imax} = \max_{j=0}^{S-1} H_j^i$$

$$h_{imin} = \min_{j=0}^{S-1} H_j^i$$

$$wh_i = \lceil \log_2(\max(h_{imax}, |h_{imin}| - 1)) \rceil + 1$$

Discussion

The results are negligible, in the 1-3% range, and in the case of pendigits they even deteriorate a bit. This deterioration should not be possible in theory if the synthesised circuits are optimal implementations of their description. This is at least evidence that the negligible results on the other datasets can be better if I can get them into a form the compiler can work better with, though I still think the main problem is with the approach itself. It seems that the truncation goes against best practices and blocks the datapath extraction process for some neurons, so certain optimizations do not get applied to them and some would be common resources do not get shared. I have not found a method to work around this limitation at this point.

Range centering

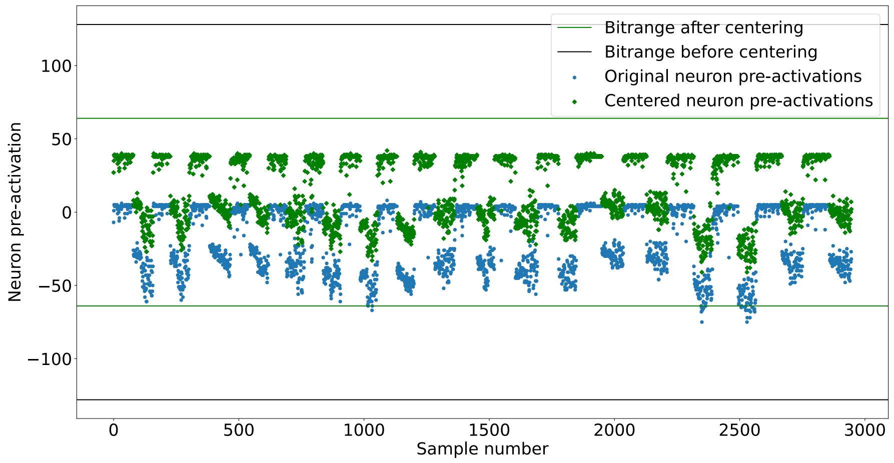


Figure 4: title here

Rationale

Unfortunately I make a mistake when initialy evaluating the results of limiting the bitwidth of neurons in the previous part. The blunder is accidentaly comparing the results with the positive-negative sums full width circuit instead of the single signed sum version that the reduced width circuits are based on. This causes me to falsely believe that reducing the bitwidths of computations is much more effective than in reality. Based on this misanderstanding I pursue finding ways to further reduce the ranges of values each neuron's total has to support. After realising the mistaken assumption I don't think those efforts were justified. The first of these is the attempt to "center" the range of values so the maximum and minimum values are at an equal distance from zero.

The range of values a neuron of the first layer can take can be significantly unbalanced. What this means is that if for example the feature the hidden neuron tracks occurs infrequently, it will equally infrequently take positive values, and this suggests that the negative values it takes can get much large in an absolute sense than the positive ones get a chance to. If the absolute minimum or maximum value is multiple times larger than the other one, let's say $h_{imax} = 10h_{imin}$, then the range of values the bitwidth of the neuron must support is up to twice as large as the one of the equivalant range balanced so it's maximum value is as far from zero as it's minimum.

Therefore by subtracting the value that lies at the middle of the range from the result, the bitwidth of the neuron's sum can be further reduced. That requires an additional constant subtraction operation for the calculation of the sum and an additional comparison operation, since the result needs to be compared to the same constant for binarization unlike being compared to zero that needs no additional hardware. However, the hope is that if the width of enough intermediate results of the adder graph and thus the number of full adders is reduced as a result it will be enough to more than make up for the cost.

Example: if the highest value the total of the neuron takes is 300 and the lowest is -100, we need a 10 bit signed integer to fit all values. If we subtract their average(100) the values will be in the range -200 to 200, so they fit in a 9 bit signed integer.

$$h_{imax} = \max_{j=0}^{S-1} H_j^i$$

$$h_{imin} = \min_{j=0}^{S-1} H_j^i$$

$$h_{imid} = \lceil \frac{h_{imax} + h_{imin}}{2} \rceil$$

$$h'_i = h_i - h_{imid}$$

$$wh'_i = \lceil \log_2 (\lceil \frac{h_{imax} - h_{imin}}{2} \rceil) \rceil + 1$$

The additional operations are only included in the design for neurons that this method decreases the bitwidth compared the one calculated from the “unbalanced” range. For the rest there is no reason to incur the overhead for no benefit so they are declared the same as they are using minimum range bit-width reduction.

Discussion

The results show that the hardware requirements of the additional operations were not covered by the reduced bitwidths and area and power demands considerably increased. Given the actual results of how much reducing bitwidths helps from above this is not surprising. In fact it seems counter-intuitive that for the winered dataset's model it lead to a 6% improvement.

Notably, this was the model that had the most benefit from the previous bitwidth reduction as well. I estimate this model's operations in the first layer lend themslves less to arithmetic optimizations compared to the rest, and thus hindering that optimization process is not as rough a sacrifice.

It also seems probable that subtracting a simpler constant than the actual value at the middle of the range h_{imid}, for example the nearest power of two, would lessen the computational burden of the additional subtraction and comparison, but not enough to be worth implementing.

Naively reducing bitwidths of intermediate results

Rationale

Given the eventual goal of applying approximate computation techniques to the adder graph of the designs a problem, mentioned previously, comes up. If the approximation, whatever it may be, gets applied separately to each neuron's sum, almost certainly blocks the datapath extraction process from performing arithmetic optimizations such as reducing and sharing common subexpressions across neurons. The result is M separate approximate adder trees. Even given that the logic reduction gained from approximate additions for each adder tree is more than enough to offset the lost benefit of intermediate result sharing, that may be an unnecessary concession.

To estimate how large the negative effect of an approximation technique not taking cross-neuron sharing into consideration could be, I reduce the bitwidths of intermediate results of the sum of each neuron. Even though this would reasonably help for a single neuron, I expect it will cause resource sharing to break. Given how large the negative effect is I can check if this is a problem than ought to be fixed before approximation can confidently be applied.

Implementation

The cumulative sum along the input features multiplied by the weight of their connection with the neuron i is calculated for each hidden neuron and for each sample of the dataset. The maximum and minimum values over all samples at each step of the cumulative sum for a given neuron is then calculated. In a similar fashion to the method described above for reducing the bitwidth of the total result of the neuron's operations, the operations are written sequentially with the result of each having it's bitwidth set based on the range of values for the equivalent step of the sum for all of the dataset.

Let $h_{i,j}$ denote the result of the pre-activation value of the i-th hidden neuron only taking into account the input features x_0 to x_j, or equivalently the value of h_i if x_{j+1} to x_{N-1} are masked to zero.

$$h_{i,j} = \sum_{k=0}^{j} x_k W1_{i,k}$$

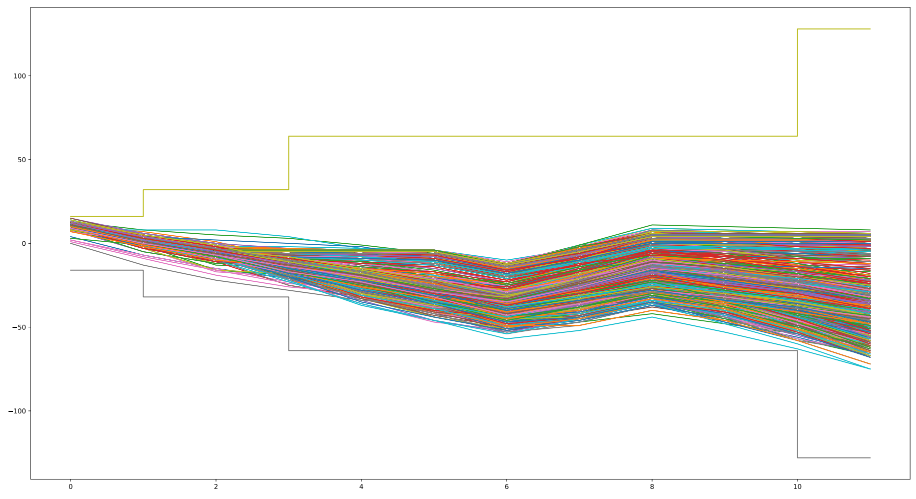


Figure 5: The intermediate sums of the sequential calculation of hidden neuron 28 for all of the Har dataset and the bit range needed to support each step

$$hmax_{i,j} = \max_{l=0}^{S-1} \sum_{k=0}^{j} D_k^l W1_{i,k}$$

$$hmin_{i,j} = \min_{l=0}^{S-1} \sum_{k=0}^{j} D_k^l W1_{i,k}$$

$$wh_{i,j} = \lceil \log_2(\max(hmax_{i,j}, |hmin_{i,j}| - 1)) \rceil + 1$$

Sometimes due to the order of additions and subtractions the width needed at a later step is less than the one of an earlier step. This is because every sample for which there would be an overflow in the earlier step with the smaller width would at some following feature underflow back in the range it supports. This has been taken into account. If a width of a subsequent operation's result is smaller, the bitwidth of the previous addition/subtraction is simply set to that smaller value.

$$wh'_{i,j} = \min_{k=j}^{N-1} wh_{i,j}$$

Consecutive operations that have the same result bitwidth are grouped together and expressed in verilog as a single sum. I have not confirmed if this does in fact affect the result of synthesis at all but it seems to conform closer to the recommendations of the best practices guide.

Discussion

The results were positive for two of the models with the smallest count of input features N, which means that this method performed better than the initial expectation. However, the optimization loss from losing shared operations shows clear scaling with the number N of input features. With more elements to sum, more common subexpressions to optimise ought to come up, so this is keeping with expectations.

At the network with the largest N, the one belonging to gasId, which has 128 sensor features, it has gotten bad enough to almost double the area and power requirements. This shows that the issue is indeed considerable when there is more than a handful of inputs, and ways to work around it shall be searched for.

Preemptive arithmetic optimization

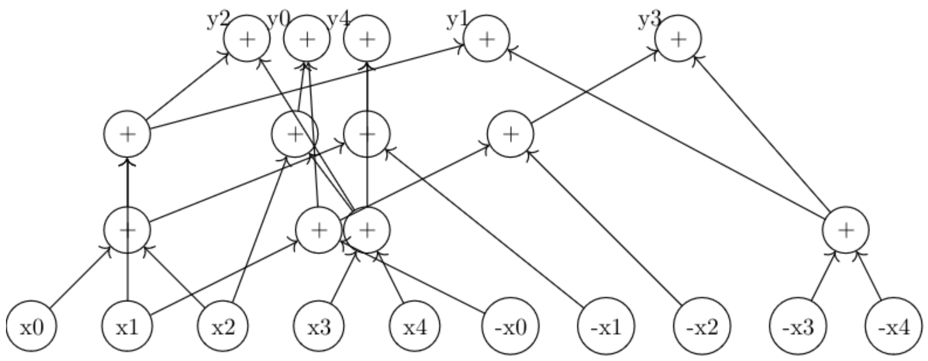


Figure 6: placeholder

Rationale

Based on the results from above I attempt to get the arithmetic expression that the design computes after synthesis, with the intent of fitting later approximation techniques to it rather than having the approximations dictate the graph of operations and forego these advantages.

Design compiler provides an "***resource and datapath extraction***" report, which quoting the user guide "analyzes the arithmetic contents of the design and provides feedback so you can improve the RTL code as needed". In this report the arithmetic operations that are performed post-optimization by each datapath block are described in the block's resource section. From this an addition / subtraction graph from input elements to layer outputs could be reconstructed with relatively straightforward parsing.

Unfortunately, the report does not provide a mapping between the symbols it uses for input and output variables of the datapath blocks and the corresponding wires in the original design. Due to this the reconstructed adder graph cannot be used to implement the network's layers before the inputs and outputs are otherwise labeled.

Conceivably a method that unfolds all intermediate steps of the calculation into full explicit sums of input variables for each output variable could be used to achieve this labeling. After all intemediate variables / nodes of the adder graph are eliminated (for example, $x = a + b; y_1 = c + x; y_2 = d - x;$ would be expanded to $y_1 = c + a + b; y_2 = d - a - b;$) the addition and subtraction operations can be mapped to a matrix with binary {-1, 1} weights. A permutation matrix that turns the original weight matrix of the layer into this reconstructed matrix is then searched for, and the permutations are assigned as the labels of the input and output elements of the adder graph.

Another path around this problem comes from the inclusion of references to the operator or operators in the original verilog design that is implemented by a given arithmetic operation mentioned in a datapath block's report. These references only point to the line of the verilog file the operator apears in, so to narrow it down with certainty the HDL file must beformated in a way that only a single arithmetic operator occurs per line.Then the variables named as operants in the report can be matched to the variables associated with the operator in the line or lines pointed to.

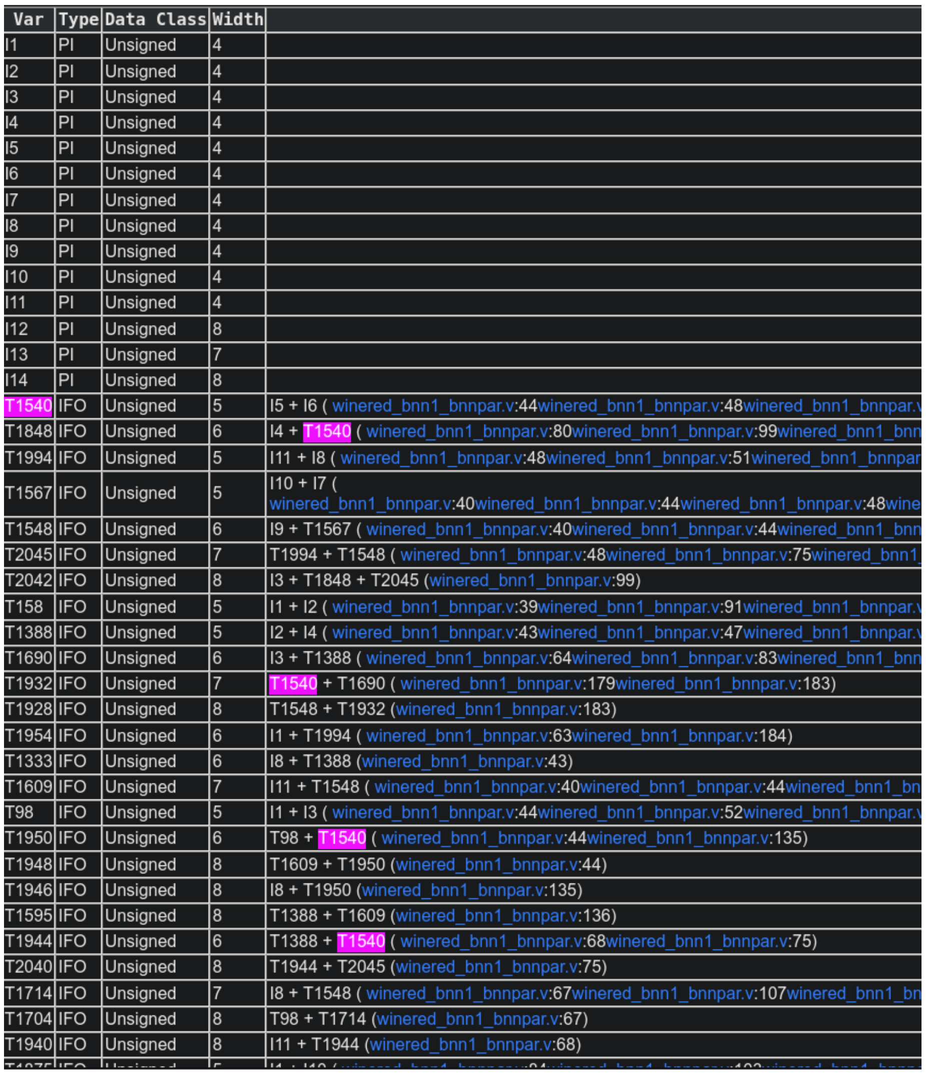
Some early attempts in this direction are made but it becomes clear that a process that involves the network to be implemented in an HDL, synthesized, reverse engineered from the synthesised result, implemented in HDL in a different way and re-synthesised would require considerably more complex orchestration than first impressions imply. Even if it panned out without any issues the speed with which modifications to the designs could be tried out would be slowed down to a degree I was not comfortable with.

As an alternative I search the literature for algorithms or heuristics that would perform an equivalent arithmetic optimization to the one Design Compiler provides. It does not appear too hopeful that a de facto standard method for such cases exists and perhaps even is the one used under the compiler's hood, so the operations found are exactly or close to the same.

Implementation

A formulation of the problem is the following: Given a list of expressions of the general form $y_i = x_0 + x_1 - x_2 - ... + x_n$ in which operants can be shared between the expressions find the minimum number of additions or subtractions that need to be performed to evaluate all expressions.

The problem turns out considerable less well-studied than initially expected. While the deceptively simple description suggests a straightforward way to answer it, it is NP-Complete difficult, more specifically in the MaxSNP family of optimization problems. As a direct result only approximate solutions are attempted. [guy] searches for exact solutions by leveraging SAT solvers, but only manages to get this to work for very small matrix sizes up to 8 x 8. Not much is else found on the exact scenario above, but a close-enough problem having to do with fac-



Var	Type	Data Class	Width	
I1	PI	Unsigned	4	
I2	PI	Unsigned	4	
I3	PI	Unsigned	4	
I4	PI	Unsigned	4	
I5	PI	Unsigned	4	
I6	PI	Unsigned	4	
I7	PI	Unsigned	4	
I8	PI	Unsigned	4	
I9	PI	Unsigned	4	
I10	PI	Unsigned	4	
I11	PI	Unsigned	4	
I12	PI	Unsigned	8	
I13	PI	Unsigned	7	
I14	PI	Unsigned	8	
T1540	IFO	Unsigned	5	I5 + I6 (winered_bnn1_bnnpar.v:44winered_bnn1_bnnpar.v:48winered_bnn1_bnnpar.
T1848	IFO	Unsigned	6	I4 + T1540 (winered_bnn1_bnnpar.v:80winered_bnn1_bnnpar.v:99winered_bnn1_bnn
T1994	IFO	Unsigned	5	I11 + I8 (winered_bnn1_bnnpar.v:48winered_bnn1_bnnpar.v:51winered_bnn1_bnnpar
T1567	IFO	Unsigned	5	I10 + I7 (winered_bnn1_bnnpar.v:40winered_bnn1_bnnpar.v:44winered_bnn1_bnnpar.v:48wine
T1548	IFO	Unsigned	6	I9 + T1567 (winered_bnn1_bnnpar.v:40winered_bnn1_bnnpar.v:44winered_bnn1_bnn
T2045	IFO	Unsigned	7	T1994 + T1548 (winered_bnn1_bnnpar.v:48winered_bnn1_bnnpar.v:75winered_bnn1_
T2042	IFO	Unsigned	8	I3 + T1848 + T2045 (winered_bnn1_bnnpar.v:99)
T158	IFO	Unsigned	5	I1 + I2 (winered_bnn1_bnnpar.v:39winered_bnn1_bnnpar.v:91winered_bnn1_bnnpar.
T1388	IFO	Unsigned	5	I2 + I4 (winered_bnn1_bnnpar.v:43winered_bnn1_bnnpar.v:47winered_bnn1_bnnpar.
T1690	IFO	Unsigned	6	I3 + T1388 (winered_bnn1_bnnpar.v:64winered_bnn1_bnnpar.v:83winered_bnn1_bnn
T1932	IFO	Unsigned	7	T1540 + T1690 (winered_bnn1_bnnpar.v:179winered_bnn1_bnnpar.v:183)
T1928	IFO	Unsigned	8	T1548 + T1932 (winered_bnn1_bnnpar.v:183)
T1954	IFO	Unsigned	6	I1 + T1994 (winered_bnn1_bnnpar.v:63winered_bnn1_bnnpar.v:184)
T1333	IFO	Unsigned	6	I8 + T1388 (winered_bnn1_bnnpar.v:43)
T1609	IFO	Unsigned	7	I11 + T1548 (winered_bnn1_bnnpar.v:40winered_bnn1_bnnpar.v:44winered_bnn1_bn
T98	IFO	Unsigned	5	I1 + I3 (winered_bnn1_bnnpar.v:44winered_bnn1_bnnpar.v:52winered_bnn1_bnnpar.
T1950	IFO	Unsigned	6	T98 + T1540 (winered_bnn1_bnnpar.v:44winered_bnn1_bnnpar.v:135)
T1948	IFO	Unsigned	8	T1609 + T1950 (winered_bnn1_bnnpar.v:44)
T1946	IFO	Unsigned	8	I8 + T1950 (winered_bnn1_bnnpar.v:135)
T1595	IFO	Unsigned	8	T1388 + T1609 (winered_bnn1_bnnpar.v:136)
T1944	IFO	Unsigned	6	T1388 + T1540 (winered_bnn1_bnnpar.v:68winered_bnn1_bnnpar.v:75)
T2040	IFO	Unsigned	8	T1944 + T2045 (winered_bnn1_bnnpar.v:75)
T1714	IFO	Unsigned	7	I8 + T1548 (winered_bnn1_bnnpar.v:67winered_bnn1_bnnpar.v:107winered_bnn1_bn
T1704	IFO	Unsigned	8	T98 + T1714 (winered_bnn1_bnnpar.v:67)
T1940	IFO	Unsigned	8	I11 + T1944 (winered_bnn1_bnnpar.v:68)
[illegible]	IFO	Unsigned	5	[illegible]

Figure 7: An example of a datapath block's extraction report. An intermediate value that is reused multiple times is highlighted.

toring similar lists of expressions of the form $y_i = x_0 \oplus x_1 \oplus ... \oplus x_n$ using the minimum possible XOR operations is actively worked on thanks to some applications in the field of cryptographic accelerators. Both belong to the shortest linear program family of problems.

I choose to try utilising Paar's factoring algorithm [paar] first. It is older than most heuristics that have been applied to the XOR factoring problem, but has the advantage of not exploiting term cancellation. Thanks to the property $x \oplus x = 0$ some optimal solutions to the XOR problem include two subexpressions containing the same term x being combined by XOR to produce a desired expression that does not include x. Heuristics developed after Paar's take advantage of this feature, and while there is a parallel between it and the term cancellation of opposites in our scenario $(x - x = 0$ or $x + -x = 0)$ I have not managed to find the adjustments needed to apply their insights to the new domain. Thus I give precedence to the more straightforward method, that directly translates to using addition in place of parity.

Paar's algorithm

It is essentially a greedy algorithm that picks the two elements that are common in the largest number of expressions each time and adds the result of their XOR to the list of elements.

Let N be the number of inputs, M be the count of expressions to evaluate, x_i be the i-th input, y_i the i-th expression and $A \in M \times N$ be the binary matrix we aim to factor. The value of $A_{i,j}$ is set to 1 if the term x_j is included in the XOR expression y_i and 0 otherwise.

For example if $N = 4$ the expression $y_i = x_0 \oplus x_2$ corresponds to the row $A_i = [1010]$.

The following steps are repeatedly applied:

1. Find the two columns $A_{:,i}$ and $A_{:,j}$ that have the bitwise AND with the largest weight. This corresponds to the columns with the largest dot product, so they can quickly be calculated by $i, j = \text{arg}\max\limits_{k>l}(A^T A)_{k,l}$. This corresponds to finding the operation between two inputs or intermediate results that occurs the most times across all expressions.
2. Append the resulting product column $A_{:,N} = A_{:,i} \wedge A_{:,j}$ to A. Increase N by one to reflect the new width of the matrix. Intuitively this translates to including $x_i \oplus x_j$ as a new intermediate result x_N to be used in farther operations.
3. Set $A'_{:,i} = A_{:,i} \wedge \neg A_{:,j}$ and $A'_{:,j} = A_{:,j} \wedge \neg A_{:,i}$. Thusly the dependence of expressions to x_i and x_j is removed when it is now covered by the inclusion of $x_i \oplus x_j$ in them.

In the end only one element in each row has value 1, and the index j of the column the only 1 of the i-th row occurs in tells as the input or intermediate result the i-th expression equals, $y_i = x_j$.

Paar's algorithm can work for a group of expressions that consist only of addition, but the expressions we have to work with include both additions and subtractions. To reconcile this issue the negative of each input element is treated as a separate input element that is added where the original would be subtracted. Given the original weight matrix W with $W_{i,j} \in -1, 1$ denoting whether element x_j is added or subtracted from the expression y_i, we construct the matrix A we pass as the initial state to the algorithm by first turning all the -1 elements to 0 and then appending the inverse of the matrix to itself, or $A = \text{max}(0, [W, -W])$.

From this process a list L of successive indices such that $L_n =$

$(i, j) \iff x_n = x_i + x_j$ is acquired, and the additions implied by them are hardcoded in the verilog description of the designs.

Extension to support subtractions

I try a slight modification to the original procedure so it can be compatible with expressions including subtractions. The way I described previously to use negative inputs with Paar's strategy often has to unnecessarily repeat calculations. For a minimal example, the expressions $y_0 = x_0 - x_1 + x_2, y_1 = x_0 + x_1 - x_2$ would result in 4 operations $(x_3 = x_0 + -x_1, y_0 = x_3 + x_2, x_4 = x_0 + x_1, y_1 = x_4 + -x_2)$, instead of the 3 needed $(x_3 = x_1 - x_2, y_0 = x_0 + x_3, y_1 = x_0 - x_3)$. This can be avoided by taking the ability to subtract instead of only add intermediate results into account.

In summary this works by using {-1, 0, 1} for the elements of the matrix A, where $A_{i,j} = -1$ when the expression for y_i contains the negative of the value x_j. Instead of counting the number of (1,1) pairs of elements for columns as before, the count of pairs of elements that are either (1,1) or (-1,-1) corresponds to the number of times the sum of the values is used, and the count of pairs of elements that are either (1,-1) or (-1,1) corresponds to the number of times the difference of the values is used. As before the operation that occurs in the most expressions is chosen as the next operation to be implemented.

Finding these two counts for all pairs of columns can still be done with a matrix multiplication like before so the new approach is not much slower.

Discussion

I expected that either:

1. The results of hardwiring the order of operations for calculating the pre-activation result of the neurons using Paar's heuristic would be considerably worse than the result of letting Design Compiler use the results of it's own optimization heuristics, since they ought to have implemented the best known ones. In this case trying an alternative to preemptively optimise the operations beforehand is most likely wasted effort because finding a competitive heuristic would be harder than parsing the results of their solution.
2. The results would have a negligible difference, because the heuristics used are related and / or the quality of results that can be expected by current methods for reasonable compute budgets hits a

certain ceiling for the various approaches. In this case applying approximation techniques to the estimated arithmetic operation graph can go ahead.

The results show that although it is not consistant across the networks, there is an improvement of 20-30% to the area and power estimates of the smallest ones. This causes suspicion, since if such an old and common algorithm performed better for some cases of the problem they would reasonably have simply used it already. I initially look for ways the original designs could be obstructing datapath extraction more than they should, but nothing jumps out to me.

There seems to be a scaling in effect where the size of the weight matrix or the number of total operations are inversely correlated with how well Paar's algorithm performs compared to the compiler's unknown solution. I hypothesise that they chose the methods they did for arithmetic optimization giving more weight to the performance on heavier workloads, where the savings are more important, or alternatively the trade-offs these methods consider apply better to heavier loads. This leaves some wiggle room for improvement in the sizes of the examined networks.

Unfortunately, the issue I am trying to address has to do with performance losses from disrupting compiler's arithmetic optimizations scaling with model size, and the relative performance of the alternative heuristic scales inversely with that size. This means that the attempted fix cannot apply to the cases that need it most, so the underlying problem remains unresolved.

The ternary take on Paar's heuristic outperforms the original by a relatively consistant ratio with exception of the Har model's network. This gives me some hope that applying more advanced heuristics used in Shortest Linear Programs modefied for this particular usecase would raise the network size threshold for which results can be improved.

Sequential evaluation of neuron’s sums