TritNet

Optimising matrix multiply computation
through scaling bit-wise logical operations

Ryan Cherian

August 2024

Contents

1 Axioms of logical operations 2
1.1 Scalar logical algebra 2
1.2 Multi-linear logical algebra 2

2 Lemmas to represent multiplication of non-bit sets as logical operations 3

3 Axioms of matrix multiplication 4

4 Matrix multiplication using only bitwise operations 5
4.1 Multiplication of binary matrices in $\{0, 1\}$ 5
4.2 Multiplication of binary matrices in $\{-1, 1\}$ 5
4.3 Multiplication of trinary matrices in $\{-1, 0, 1\}$ 5
4.4 Multiplication of trinary-binary matrices 6

5 Extension 7

6 Consequences 8
6.1 Problem formulation 8
6.2 Options 8

1 Axioms of logical operations

1.1 Scalar logical algebra

Note that throughout this derivation, we abide the notation that Greek letters represent bits, and all other variables (incl. binary variables outside $\{0, 1\}$, e.g. in $\{-1, 1\}$) are Roman. The axioms in sum are:

$\forall \alpha, \beta \in \{0, 1\}$

Axiom 1.1 (AND): $\alpha \wedge \beta = \alpha\beta$

Axiom 1.2 (OR): $\alpha \vee \beta = \alpha + \beta - \alpha\beta$

Axiom 1.3 (NOT): $\bar{\alpha} = 1 - \alpha$

Table 1: Truth tables

α	β	$\boldsymbol{\alpha\beta}$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: AND

α	β	$\boldsymbol{\alpha + \beta - \alpha\beta}$
0	0	0
0	1	1
1	0	1
1	1	1

Table 2: OR

α	$\mathbf{1} - \boldsymbol{\alpha}$
0	1
1	0

Table 1: NOT

1.2 Multi-linear logical algebra

A population count (`_popc()`) is equivalent to a *full* tensor contraction (one form of multi-linear reduction) in integer or floating-point arithmetic - i.e. the sum of all elements of an n-rank tensor.

Axiom 1.4 (Population Count): $\texttt{popc}(\boldsymbol{\alpha}) := \sum_i \alpha_i$

Thus, if a tensor can be represented in first-rank (vector) form, and as a concatenation of strictly Boolean operations, it's full contraction can be represented as a pop. count. For instance:

$$\sum_{i,j} \alpha_{i,j} \vee \beta_{i,j} = \texttt{popc}(\boldsymbol{\alpha} \vee \boldsymbol{\beta})$$

$$\sum_{k} \alpha_{i,k} \wedge \beta_{k,j} = \texttt{popc}(\boldsymbol{\alpha}_i \wedge \boldsymbol{\beta}_j) \quad = \boldsymbol{\alpha}_i \cdot \boldsymbol{\beta}_j$$

Where the last equality represents summations of the dot product of Boolean vectors. In other words, the *matrix multiplication* of Boolean matrices.

2 Lemmas to represent multiplication of non-bit sets as logical operations

Lemma 1 (OR): $\alpha \vee \beta = \alpha\beta! + \beta$

Proof.

$$\begin{aligned}\alpha \vee \beta &= \alpha + \beta - \alpha\beta \\ &= \alpha(1-\beta) + \beta = \alpha\beta! + \beta\end{aligned}$$

□

Lemma 2 (XOR): $\alpha \veebar \beta = \alpha\beta! + \beta\alpha! = \alpha + \beta - 2\alpha\beta$

Proof.

$$\begin{aligned}\alpha \veebar \beta &= \alpha\beta! + \beta\alpha! \qquad \text{(truth table)} \\ &= \alpha(1-\beta) + \beta(1-\alpha) = \alpha + \beta - 2\alpha\beta\end{aligned}$$

□

Lemma 3 (Transform): $a \in \{-1, 1\} \mapsto \alpha \in \{1, 0\}$ through $\alpha = \frac{1-a}{2}$

Proof. Simple linear transformation:

$$\forall\ a \in \{-1, 1\}\ \exists\ \alpha = \frac{1-a}{2} : \alpha \in \{1, 0\}$$

□

Lemmas 4.x Multiplication of binary variables:

Lemma 4.0 Binary multiplication in $\alpha, \beta \in \{0, 1\}$: $\alpha\beta = \alpha \wedge \beta$

(This is Axiom 1)

Lemma 4.1 Binary multiplication of $\forall a, b \in \{-1, 1\}$: $ab = 1 - 2(\alpha \veebar \beta)$

Proof.

$$\begin{aligned}\textbf{Lemma 3} \implies \qquad & ab = (1-2\alpha)(1-2\beta) = 1 - 2\alpha - 2\beta + 4\alpha\beta \\ & \quad = 1 - 2(\alpha + \beta - 2\alpha\beta) \\ \textbf{Lemma 2} \implies \qquad & \quad = 1 - 2(\alpha \veebar \beta)\end{aligned}$$

□

Lemma 4.2: Binary multiplication $\alpha \in \{0, 1\}$ with $b \in \{-1, 1\}$: $\alpha b = \alpha - 2\alpha\beta = \alpha \veebar \beta - \beta$

Proof.

$$\begin{aligned}\textbf{Lemma 3} \implies \qquad & \alpha b = \alpha(1-2\beta) \quad \forall \alpha \in \{0, 1\},\ b \in \{-1, 1\} \\ & \quad = \alpha - 2\alpha\beta \qquad = \alpha - 2(\alpha \wedge \beta) \\ \textbf{Lemma 2} \implies \qquad & \quad = \alpha \veebar \beta - \beta\end{aligned}$$

□

3 Axioms of matrix multiplication

General matrix multiplication $\mathbf{Z} = \mathbf{WA}$ is defined as follows:

$$\textbf{Axiom 2.1:} \qquad \exists \ \mathbf{Z} = \mathbf{WA} : Z_{i,j} = \sum_k W_{i,k} A_{k,j} \qquad \forall \ W_{i,k}, A_{k,j}, Z_{i,j} \in \mathbb{R}$$

If using NVIDIA Tensor Cores, we can define two new binary matrix multiplication types:

$$\textbf{Axiom 2.2:} \qquad \exists \ \mathbf{Z} = \omega * \alpha : Z_{i,j} = \sum_k (\omega_{i,k} \wedge \alpha_{k,j}) \qquad \forall \ \omega_{i,k}, \alpha_{k,j} \in \{0,1\}, \ Z_{i,j} \in [0.. \, ..k]$$

$$\textbf{Axiom 2.3:} \qquad \exists \ \mathbf{Z} = \omega \star \alpha : Z_{i,j} = \sum_k (\omega_{i,k} \veebar \alpha_{k,j}) \qquad \forall \ \omega_{i,k}, \alpha_{k,j} \in \{0,1\}, \ Z_{i,j} \in [0.. \, ..k]$$

Note that NVIDIA's cores each operate on a maximum of $(8, 128, 8)$ dimensions.

4 Matrix multiplication using only bitwise operations

4.1 Multiplication of binary matrices in $\{0, 1\}$

$$Z_{i,j} = \sum_k \omega_{i,k}\alpha_{k,j} \qquad \forall\ \omega_{i,k}, \alpha_{k,j} \in \{0,1\}$$

Axiom 1.1 $\implies$
$$Z_{i,j} = \sum_k \omega_{i,k} \wedge \alpha_{k,j}$$

Axioms 2.1, 2.2 $\implies$
$$\therefore \mathbf{Z} = \boldsymbol{\omega}\boldsymbol{\alpha} = \boldsymbol{\omega} * \boldsymbol{\alpha}$$

4.2 Multiplication of binary matrices in $\{-1, 1\}$

$$Z_{i,j} = \sum_k W_{i,k}A_{k,j} \qquad \forall\ W_{i,k}, A_{k,j} \in \{-1,1\},\ Z_{i,j} \in k \cdot [W \times A]$$

Lemma 4.1 $\implies$
$$= \sum_k 1 - 2(\omega_{i,k} \veebar \alpha_{k,j})$$
$$= k - 2\sum_k (\omega_{i,k} \veebar \alpha_{k,j})$$

Axioms 2.1,2.3 $\implies$
$$\therefore\ \mathbf{Z} = \mathbf{WA} = \mathbf{K} - 2(\boldsymbol{\omega} \star \boldsymbol{\alpha}) \qquad \forall\ K_{i,j} \in \{k\}$$

So for a (m, k, n) matrix multiply, calculated compactly through bitwise operations on N-bit words:

```
    Z[i][j] += N - 2*_popc(W[i][k] ^ A[k][j]);

    //or equivalentally
    Z[i][j] += N - (_popc(W[i][k] ^ A[k][j])<<1);

    //or when utilising NVIDIA Tensor Core capability:
    Z[i][j] = n*k - 2*(bmma_PopXor(W,A))[i][j];        //_popc(W[i][k] ^ A[k][j])
```

4.3 Multiplication of trinary matrices in $\{-1, 0, 1\}$

$$W_{i,j}, A_{i,j} \in \{-1,0,1\}\ \forall\ i,j\ \in \mathbb{N} : \mathbf{Z} = \mathbf{WA}$$
$$\exists\ \omega^o_{i,j}, \alpha^o_{i,j} \in \{0,1\},\ W'_{i,j}, A'_{i,j} \in \{-1,1\} : \mathbf{W} = \boldsymbol{\omega}^o \odot \mathbf{W}',\ \mathbf{A} = \boldsymbol{\alpha}^o \odot \mathbf{A}'$$

$$\therefore \mathbf{Z} = (\boldsymbol{\omega}^o \odot \mathbf{W}')(\boldsymbol{\alpha}^o \odot \mathbf{A}')$$

Axiom 1 $\implies$
$$Z_{i,j} = \sum_k W_{i,k}A_{k,j} = \sum_k \omega^o_{i,k}W'_{i,k}\alpha^o_{k,j}A'_{k,j}$$
$$= \sum_k \omega^o_{i,k}\alpha^o_{k,j}W'_{i,k}A'_{k,j}$$

Lemma 4 $\implies$
$$= \sum_k \omega^o_{i,k}\alpha^o_{k,j}(1 - 2(\omega'_{i,k} \veebar \alpha'_{k,j}))$$
$$= \sum_k \omega^o_{i,k}\alpha^o_{k,j} - 2\sum_k \omega^o_{i,k}\alpha^o_{k,j}(\omega'_{i,k} \veebar \alpha'_{k,j})$$

This gives the calculation for each element. Thus you can stop here for the operation per kernel thread:

```
Z[i][j] += _popc(W0[i][k] & A0[k][j]) - 2*_popc(W0[i][k] & A0[k][j] & (W1[i][k] ^ A1[k][j]));
```

Or if utilising Tensor cores, continue by expanding the XOR operation using Lemma 2, then simplify:

$$\textbf{Lemma 2} \implies = \sum_k \omega^o_{i,k}\alpha^o_{k,j} - 2\sum_k \omega^o_{i,k}\alpha^o_{k,j}(\overline{\omega'_{i,k}}\alpha'_{k,j} + \omega'_{i,k}\overline{\alpha'_{k,j}})$$
$$= \sum_k \omega^o_{i,k}\alpha^o_{k,j} - 2\sum_k \omega^o_{i,k}\alpha^o_{k,j}\overline{\omega'_{i,k}}\alpha'_{k,j} - 2\sum_k \omega^o_{i,k}\alpha^o_{k,j}\omega'_{i,k}\overline{\alpha'_{k,j}}$$
$$= \sum_k \omega^o_{i,k}\alpha^o_{k,j} - 2\sum_k (\omega^o_{i,k}\overline{\omega'_{i,k}})(\alpha^o_{k,j}\alpha'_{k,j}) - 2\sum_k (\omega^o_{i,k}\omega'_{i,k})(\alpha^o_{k,j}\overline{\alpha'_{k,j}})$$

$$\textbf{Axioms 1,2} \implies \therefore \mathbf{Z} = \mathbf{WA} = \boldsymbol{\omega}^o\boldsymbol{\alpha}^o - 2(\boldsymbol{\omega}^o \odot \overline{\boldsymbol{\omega}'})(\boldsymbol{\alpha}^o \odot \boldsymbol{\alpha}') - 2(\boldsymbol{\omega}^o \odot \boldsymbol{\omega}')(\boldsymbol{\alpha}^o \odot \overline{\boldsymbol{\alpha}'})$$
$$= \boldsymbol{\omega}^o\boldsymbol{\alpha}^o - 2\mathbf{PQ} - 2\mathbf{RS}$$

This translates to:

```
Z = bmma_PopAND(W0,A0) - 2*(bmma_PopAND(W0(!W1), A0A1)) - 2*(bmma_PopAND(W0W1, A0(!A1)));
```

4.4 Multiplication of trinary-binary matrices

$$W_{i,j} \in \{-1,0,1\}, A_{i,j} \in \{-1,1\} \; \forall \; i,j \; \in \mathbb{N} : \mathbf{Z} = \mathbf{WA}$$
$$\exists \; \omega^+_{i,j}, \omega^-_{i,j} \in \{0,1\} : \mathbf{W} = \boldsymbol{\omega}^+ - \boldsymbol{\omega}^-$$

$$\therefore \mathbf{Z} = (\boldsymbol{\omega}^+ - \boldsymbol{\omega}^-)\mathbf{A}$$
$$\textbf{Axiom 1} \implies \quad Z_{i,j} = \sum_k W_{i,k}A_{k,j} = \sum_k (\omega^+_{i,k} - \omega^-_{i,k})A_{k,j}$$
$$= \sum_k \omega^+_{i,k}A_{k,j} - \sum_k \omega^-_{i,k}A_{k,j}$$
$$\textbf{Lemma 4} \implies \quad = \sum_k (\omega^+_{i,k} \veebar \alpha_{k,j} - \alpha_{k,j}) - \sum_k (\omega^-_{i,k} \veebar \alpha_{k,j} - \alpha_{k,j})$$
$$= \sum_k \omega^+_{i,k} \veebar \alpha_{k,j} - \sum_k \omega^-_{i,k} \veebar \alpha_{k,j}$$
$$\textbf{Axiom 2.3} \implies \quad \therefore \mathbf{Z} = \mathbf{WA} = (\boldsymbol{\omega}^+ \star \boldsymbol{\alpha}) - (\boldsymbol{\omega}^- \star \boldsymbol{\alpha})$$

Thus you can calculate each element using the following operation per kernel thread:

```
Z[i][j] += _popc(W+[i][k] ^ A[k][j]) - _popc(W-[i][k] ^ A[k][j]);
```

Or calculate the whole matrix using tensor cores, with the following:

```
Z = bmma_PopXOR(W+,A) - bmma_PopXOR(W-,A);
```

5 Extension

Note that this method is **extendable**. Consider the following split:

$$\in \{2, 1, 0, -1\} = abcd \forall a\{0, 1\} b\{1, 1\} c\{1, 2\} d\{1, -1\}$$

With this we can then simplify the multiplication of the two quaternary variables like so:

$$abcd * pqrs = ap * bq * cr * ds = ...$$

However, we then require 4 bits to hold each quaternary variable - which could easily have been held in 2 bits. So this method is not feasible beyond multiplications of sets with 3 elements.

6 Consequences

6.1 Problem formulation

Assume memory-bound, not compute-bound. i.e. our problem requires two things: loading/saving stored parameters, and performing operations on the parameters.

Consider a factory with one worker doing job X, and one working doing job Y. Both workers are not always working (max efficiency). One worker has significant idle time.

Here, the operations worker (cuda core) has idle time - i.e. the problem is memory-bound.

Since the hardware is fixed, we want to adapt our algorithm such that we minimise number of memcalls.

We want each call to carry as many parameters as possible. (32 parameters per byte). Assume each memcall occupies full bandwidth of bus eg. 32 bytes/call (=> 32 parameters/call).

Thus we originally quantised the network into bits. Now same, but for trits.

two options: store as two bits s.t. 8bit word has 4 trits - 01,00,10,11 store s.t. 2 8bit words have 8 trits. 0101..., 0110...

im doing the second option since its easier to think about, but is it actually quicker than storing trits contiguously as 2 bits? Well if using tensor cores, yes definitely since that loads contiguous chunks at a time. If doing on a thread-wise basis...? Yes, definitely better to store as separate matrices, cus cant do bitwise operations if theyre stored as pairs!!! duh! that took too long to realise lol.

6.2 Options

So overall we have 4 options:

- thread-wise (W+ - W-)(A+ - A-) => popc(WA) - popc(WA) - popc(WA) + popc(WA)
- thread-wise (W0W1)(A0A1) => `popc(AB) - 2*popc(AB(C^D))`
- tensor core (W+ - W-)(A+ - A-) => A.C - B.C - A.D + B.D
- tensor core (W0W1)(A0A1) => A.B - A!C.BD - AC.B!D

(Also note that a-b cannot be repesented as)

option 1 is rubbish because it requires 3 summations of 4 popcounts. its also rubbish because writing the output after the activation is much more difficult when both Z+ and Z- carry *independent* information, whereas in option 2 Z0 and Z1 are *dependent*, so we can perform the memset in one if statement. Also both option 1 and 2 require the same number of matrices.

But how does option 2 compare to the best out of 3 and 4?

Well, from the tensor core options, option 4 requires 6 matrices, which is 2 more than option 3... but we can actually keep it at 4 matrices and split the kernel into two tasks: elementwise mult and matrix mult. Then both options have the same memory load time. Then their complexities are:

3: $4(n^3(M+A)) + 3 * An^2$ vs 4: $3(n^3(M+A)) + 4(Mn^2) + 2 * An^2$ where the M and A represent cost of multiply and cost of add. Here multiply actually costs more than add, since our multiplication is simply the AND or XOR operations! We can denote this through some scaling factor A = kM.

However, we must also account for the overhead time to run two kernels (the hamadard and then the matmul)!

But this is all just compute time. Are we forgetting memory time? Well both 3 and 4 require the same memory pull time since we can keep option 4 down to 4 matrices.

Note that it is possible to access `c_frag` from the registers before storing it to global memory!! In fact we can discard `c_frag` completely, and just write the processed values.

So finally it seems option 4 is again faster than 3.

But overall is 2 faster than 4? Not sure.

The tricky thing with tensor cores is that I dont know how we're supposed to feed in the data... in `uint1b_t` or in `experimentail::precision::b1`?