# Computable Compressed Matrices Suplementary Material

Crysttian A. Paixão  Flávio Codeço Coelho

March 27, 2013

## 1 Introduction

In this supplementary material we will explore basic arithmetic operations (addition, subtraction, division and multiplication) over the elements of bit-string compressed arrays. We will consider integer matrices compressed via both the SM and the VLB methods.

In these operations, the most important aspect is the bit-length of the result in relation to those of the operands. As the operations are all done in binary, when the result bit-length increases, the resulting matrix will require more space to store and the operation in itself gets more complex, as in place operations are not possible. In the following examples we will explore these kinds of operations.

### 1.1 Operations with Scalars

**Addition** In base 2 the carry over behavior is the same as we observer when operating with decimals, only the base is different. Adding $1 + 1$ results in 0 and a carry over of 1, this is similar to the decimal sum $5 + 5$ where we also have a carry over of 1. Other single digit additions in base 2 are: $1 + 0 = 1$, $0 + 1 = 1$, and $0 + 0 = 0$. Consider the following addition $n_1 + n_2$ where $n_1 = 7$ and $n_2 = 10$. This addition is illustrated in table 1 in both decimal and binary. Note that when we add $1 + 1$, a carry over is generated to be added to bit on the left. In total we generate three bits in carry over.

In Table 2 we show the addition of $7 + 7$ which generates a 4 bits long number. In integer additions, the results may, in the very least, be as long

(in bits) as the greatest operand (for example $1 + 2 = 3$), but is frequently longer.

Table 1: Comparing binary and decimal sum $(7 + 10)$.

| | Decimal | Binary | bit-length |
|---|---|---|---|
| Carry Over | | *11100* | |
| $n_1$ | 7 | 111 | 3 |
| $n_2$ | 10 | 1010 | 4 |
| Total | 17 | 10001 | 5 |

Table 2: Comparacao da soma em base decimal e binaria.

| | Decimal | Binario | Bit-length |
|---|---|---|---|
| Carry | | *1110* | |
| $n_1$ | 7 | 111 | 3 |
| $n_1$ | 7 | 111 | 3 |
| Total | 14 | 1110 | 4 |

**Subtraction** Depending of the number involved in this operation, the result may be negative. This fact alone generates the need for an extra bit to represent the sign (+ or −). The usual way to achieve this is to use the two's complement [1] representation which has the advantage of making the addition subtraction and multiplication operation the same as those for unsigned binary numbers. This algorithm is what is actually implemented in CPUs. To exemplify let's subtract 7 from 10. To make use of the same algorithm of the sum , we represent the operation as $10 + (-7)$. First we need to convert the operands to two's complement representation (C2):

- The numbers will require an extra bit in C2. Thus 7 which is 3 bits long in binary, will require 4 bits in C2. As 7 will be added to 10, the operands must be represented by 5 bits (4 bits to represent 10 plus one for the signal). Therefore 7 becomes 00111 (in one's complement).
- Now the bits in 7 are flipped, going from 00111 to 11000.
- to the flipped number we add 1: $11000 + 1 = 11001$, which is the C2 representation of $-7$

Once completed the conversion, we add the numbers (see table 3). Both operands require 5 bits because of C2. Most importantly, the left most bit generated by the carry over, is discarded. That happens because to operate in C2, the operands must have the same bit-length so any overflow in the result must be discarded.

This operation would be possible without the use of C2 representation as 10 is greater than 7, and no signed integer is involved (see table 4 for this).

Table 3: Two's complement addition(subtraction) in decimal and binary.

| | Decimal | C2 Binary | Bit-length |
|---|---|---|---|
| Carry over | | *110000* | |
| $n_2$ | 10 | 01010 | 5 |
| $+(-n_1)$ | +(-7) | 11001 | 5 |
| Total | 3 | ~~1~~00011 | 5 |

Table 4: Standard binary subtraction.

| | Decimal | Binary | Bit-length |
|---|---|---|---|
| Carry over | | *0111* | |
| $n_2$ | 10 | 1010 | 4 |
| $-n_1$ | -7 | 111 | 3 |
| Total | 3 | 11 | 2 |

In conclusion, in the worst case subtraction will require the same number of bits of the greatest operand, regardless of the use of C2 representation. As an example, consider the subtraction $7 - 1$ (table 5).

Table 5: Binary subtraction

| | Decimal | Binary | Bit-length |
|---|---|---|---|
| Carry over | | *000* | |
| $n_1$ | 7 | 111 | 3 |
| $-n_3$ | -1 | 1 | 1 |
| Total | 6 | 110 | 3 |

**Multiplication** If multiplication is thought of as a series of sums, we can apply much the same techniques. For example: $2 \times 3 = 3 + 3 = 2 + 2 + 2$. Table 6 describes a simple multiplication.

Table 6: Binary Multiplication

| | Decimal | Binary | Bit-length |
|---|---|---|---|
| $n_2$ | 10 | 1010 | 4 |
| $n_1$ | $\times 7$ | 111 | 3 |
| | | 1010 | |
| | | 1010 | |
| | | 11110 | |
| | | 1010 | |
| Total | 70 | 1000110 | 7 |

In multiplication the behavior of the bit-length is different from the sum and subtraction. In the worst case the product has a bit-length which is the sum of the bit-lengths of the factors. For example, in the product $7 \times 7$, the product is 6 bits long: 49 which in binary is 110001.

If a signed integer is involved in the multiplication, we need to use C2 representation. To illustrate this case let's calculate $10 \times -7$. in C2, this operation becomes $01010 \times 11001$. Table 7 contains the details. The basic difference is that the sign bits (in bold), positioned to the left, are not involved in the operation. Only in the end they are used to determine the sign of the product: 0 meaning positive and 1 negative. In the worst case the bit-length of the product is the sum of the bit-lengths of the factors plus 1 due to the sign.

**Division** Division, in contrast to multiplication, requires successive subtractions until a remainder is reached which may or may not be zero. To illustrate the procedure in binary, table 8 describes the division of 50 by 10. Note the successive binary subtractions. The bit-length of the quotient is, in the worst case, equal to the bit-length of the dividend when this is longer than that of the divisor. With each subtraction, we add 1 to the quotient which is initially 0. The process ends when the remainder is 0 or less than the divisor.

When the division involves a signed operand, we do the same as with the multiplication, the operations is executed on the unsigned operands and the

Table 7: Multiplication involving a signed integer.

| | Decimal | Binary | Bit-length |
|---|---|---|---|
| $n_2$ | 10 | **0**1010 | 5 |
| $\times -1 n_1$ | $\times -7$ | **1**1001 | 5 |
| | | 1010 | |
| | | + 0000 | |
| | | 01010 | |
| | | + 0000 | |
| | | 001010 | |
| | | + 1010 | |
| Total | -70 | **1**1011010 | 8 |

sign is applied at the end.

Table 8: Binary division as a series of subtractions.

| | Decimal | Binary | Decimal quotient | Binary quotient |
|---|---|---|---|---|
| $n_4$ | 50 | 110010 | 0 | 000 |
| $n_2$ | -10 | 001010 | +1 | +001 |
| Remainder | 40 | 101000 | 1 | 001 |
| $n_2$ | -10 | 001010 | +1 | +001 |
| Remainder | 30 | 011110 | 2 | 010 |
| $n_2$ | -10 | 001010 | +1 | +001 |
| Remainder | 20 | 010100 | 3 | 011 |
| $n_2$ | -10 | 001010 | +1 | +001 |
| Remainder | 10 | 001010 | 4 | 100 |
| $n_2$ | -10 | 001010 | +1 | +001 |
| Remainder | 0 | 000000 | 5 | 101 |

So far we have examined the 4 fundamental arithmetic operations. In summary the implication for memory allocation of the results are the folowing, in the worst case scenarios:

- Addition: requires 1 extra bit above the bit-length of the greatest operand;
- Subtraction: Requires the same number of bits as the greatest operand;
- Multiplication: Requires the sum of the bit-lenghts of the operands;

- Division: Requires the same bit-length as the dividend;

Figure 1 shows the bit-lengths of results of the four operations with integer operands up to 8 bits in length. The conclusions listed above, are visually emphasized in the figures.

When operating with signed integers, an extra bit is used for the sign.

The operations on single numbers (scalars) as exposed above are a simplification of the actual operations taking place on the bitstrings as we compute with compressed matrices. Additional details will be provided below, when we discuss the operation with matrices. Note that we are restricting the example to operations which generate integer results. Handling of float operations and compression will be the subject of a subsequent paper.

Figure 1: Bit-length of the results for each of the four operations. The axes represent the values of the operands.

## 1.2 Operation on Bitstrings

To illustrate operations with bitstrings, consider the matrices $A$ an $B$, both $3 \times 4$:

$$A = \begin{bmatrix} 1 & 3 & 5 & 8 \\ 12 & 14 & 6 & 9 \\ 3 & 7 & 10 & 11 \end{bmatrix} \tag{1}$$

e

$$B = \begin{bmatrix} 1 & 3 & 7 & 9 \\ 1 & 3 & 14 & 15 \\ 20 & 30 & 2 & 1 \end{bmatrix} \tag{2}$$

After bitstring compression by the SM method, they become $3 \times 1$. They are show in decimal and binary from in 3 and 4. The elements in $A$ and $B$ are 4 and 5 bits long, respectively (the "supreme minimum"). The ellipses ("...") in the binary matrix correspond the extra zeros to the left. Since the strings are written into 64 bits integers, sometimes we end up with unused bits to the left.

$$AA = \begin{bmatrix} 4952 \\ 52841 \\ 14251 \end{bmatrix} = \begin{bmatrix} \ldots 0001001101011000 \\ \ldots 1100111001101001 \\ \ldots 0011011110101011 \end{bmatrix} = \begin{bmatrix} \ldots \underbrace{0001}_{1}\underbrace{0011}_{3}\underbrace{0101}_{5}\underbrace{1000}_{8} \\ \ldots \underbrace{1100}_{12}\underbrace{1110}_{14}\underbrace{0110}_{6}\underbrace{1001}_{9} \\ \ldots \underbrace{0011}_{3}\underbrace{0111}_{7}\underbrace{1010}_{10}\underbrace{1011}_{11} \end{bmatrix} \quad (3)$$

e

$$BB = \begin{bmatrix} 36073 \\ 36303 \\ 686145 \end{bmatrix} = \begin{bmatrix} \ldots 000010001100111010 01 \\ \ldots 000010001100111011 11 \\ \ldots 101001111000010000 01 \end{bmatrix} = \begin{bmatrix} \ldots \underbrace{00001}_{1}\underbrace{00011}_{3}\underbrace{00111}_{7}\underbrace{01001}_{59} \\ \ldots \underbrace{00001}_{1}\underbrace{00011}_{3}\underbrace{01110}_{14}\underbrace{01111}_{15} \\ \ldots \underbrace{10100}_{20}\underbrace{11110}_{30}\underbrace{00010}_{2}\underbrace{00001}_{1} \end{bmatrix} \quad (4)$$

Consider now just the bitstring stored in the first line of each matrix ($AA$ and $BB$). To recover the original elements we need to now the bit-lengthof the blocks of memory containing them, for the SM method they are all the same length. To obtain the values in an efficient way, we will use a binary mask. The mask will recover the first and third blocks at the same time to save time. The mask is stored as a bitstring the same length as those storing the matrix elements. The mask for the first and third elements of matrix $AA$ is depicted in 5. To recover the second and fouth elements, we apply the mask shown on 6. For Matrix $BB$, see 7 and 8 for the respective masks.

$$\text{Matrix AA's mask for positions 1 and 3} = \ldots 0000111100001111 \quad (5)$$

$$\text{Matrix AA's mask for positions 2 and 4} = \ldots 1111000011110000 \quad (6)$$

$$\text{Matrix BB's mask for positions 1 and 3} = \ldots 00000111110000011111 \quad (7)$$

$$\text{Matrix BB's mask for positions 2 and 4} = \ldots 11111000001111100000 \quad (8)$$

Once defined the mask we can apply it to matrices $A$ and $B$. We use the mask by applying the boolean function AND. Let tempA and tempB the recovered elements. The recovering process is illustrated below. Note that only the positions where the mask is 1 are retained.

$$\begin{aligned} A[1,1] &= \ldots 0001|0011|0101|1000 \\ \text{mask A} &= \ldots 0000|1111|0000|1111 \\ tempA &= \ldots 0000|0011|0000|1000 \end{aligned}$$

$$\begin{aligned} B[1,1] &= \ldots 00001|00011|00111|01001 \\ \text{mask B} &= \ldots 00000|11111|00000|11111 \\ tempB &= \ldots 00000|00011|00000|01001 \end{aligned}$$

**Addition** Suppose we want to add both matrices and store it in a matrix $S$. for that the blocks of tempA and tempB must have the same length. As for the addition we need, in the worst case, an extra bit, the block length of the resulting bitstring will be 6. The string from each matrix, padded on the left towards this new length, are shown in equations 9 and 10.

$$tempA = \ldots \mathbf{00}0000\mathbf{00}0011\mathbf{00}0000\mathbf{00}1000 \tag{9}$$

$$tempB = \ldots \mathbf{0}00000\mathbf{0}00011\mathbf{0}00000\mathbf{0}01001 \tag{10}$$

Now that the blocks are matched in length we can perform the operation $S[1,1] = tempA + tempB$. The operation is repeated for the second and fourth regions. these operations can be done in parallel and the result, $S[1,1]$, is already in compressed form.

$$\begin{aligned} tempA &= \ldots 000001|000011|000101|001000 \\ tempB &= \ldots 000001|000011|000111|001001 \\ S[1,1] &= \ldots 000010|000110|001100|010001 \end{aligned}$$

We can see below the same operation performed for bitstrings $S[2,1]$ and $S[3,1]$.

$$
\begin{aligned}
tempA &= \ldots 001100|001110|000110|001001 \\
tempB &= \ldots 000001|000011|001110|001111 \\
S[2,1] &= \ldots 001101|010001|010100|011000
\end{aligned}
$$

$$
\begin{aligned}
tempA &= \ldots 000011|000111|001010|001011 \\
tempB &= \ldots 010100|011110|000010|000001 \\
S[3,1] &= \ldots 010111|100101|001100|001100
\end{aligned}
$$

Therefore, as the final result of the operation $S = A + B$ we have:

$$
\begin{aligned}
S = \begin{bmatrix} 549649 \\ 274889 \\ 6181644 \end{bmatrix} &= \begin{bmatrix} \ldots 000010001100011000010001 \\ \ldots 000001000011000111001001 \\ \ldots 010111100101001100001100 \end{bmatrix} \\
&= \begin{bmatrix} \ldots \underbrace{000010}_{2}\underbrace{000110}_{6}\underbrace{001100}_{12}\underbrace{010001}_{17} \\ \ldots \underbrace{001101}_{13}\underbrace{010001}_{17}\underbrace{010100}_{20}\underbrace{011000}_{24} \\ \ldots \underbrace{010111}_{23}\underbrace{100101}_{37}\underbrace{001100}_{12}\underbrace{001100}_{12} \end{bmatrix}
\end{aligned} \tag{11}
$$

**Subtraction** To perform the subtraction, we need to use C2 notation. Supose we need to calculate $D = B - A$. This operation can be done in the same way, by converting it to an addition: $D = B + (-A)$, where $-A$ will be converted to C2 (equation 12). Note the elements will be calculated with six digits.

$$
\begin{aligned}
-A = \begin{bmatrix} 4952 \\ 52841 \\ 14251 \end{bmatrix} &= \begin{bmatrix} \ldots 111111111101111011111000 \\ \ldots 110100110010111010110111 \\ \ldots 111101111001110110110101 \end{bmatrix} \\
&= \begin{bmatrix} \ldots \underbrace{111111}_{-1}\underbrace{111101}_{-3}\underbrace{111011}_{-5}\underbrace{111000}_{-8} \\ \ldots \underbrace{110100}_{-12}\underbrace{110010}_{-14}\underbrace{111010}_{-6}\underbrace{110111}_{-9} \\ \ldots \underbrace{111101}_{-3}\underbrace{111001}_{-7}\underbrace{110110}_{-10}\underbrace{110101}_{-11} \end{bmatrix}
\end{aligned} \tag{12}
$$

Now we apply the masks as before, but before we need to convert to C2 as well, in order to perform the addition.

To exemplify, let's examine closely the operations $B[1,1] + (-A[1,1])$, $B[2,1] + (-A[2,1])$ and $B[3,1] + (-A[3,1])$. We use tempB to store each line of the matrix B converted to C2. The numbers 1 marked in the bitstring, are overflows and must be removed. The result is shown below:

$$\begin{aligned} tempB &= \ldots 000001|000011|000111|001001 \\ +(-A[1,1]) &= \ldots 111111|111101|111011|111000 \\ D[1,1] &= \ldots \underline{1}000000|\underline{1}000000|\underline{1}000010|\underline{1}000001 \end{aligned}$$

$$\begin{aligned} tempB &= \ldots 000001|000011|001110|001111 \\ +(-A[2,1]) &= \ldots 110100|110010|111010|110111 \\ D[2,1] &= \ldots 110101|110101|\underline{1}001000|\underline{1}000110 \end{aligned}$$

$$\begin{aligned} tempB &= \ldots 010100|011110|000010|000001 \\ +(-A[3,1]) &= \ldots 111101|111001|110110|110101 \\ D[3,1] &= \ldots \underline{1}010001|\underline{1}110111|\underline{1}111000|110110 \end{aligned}$$

After the removal of the overflows, matrix $D$ becomes:

$$D = \begin{bmatrix} 129 \\ 14111238 \\ 4685366 \end{bmatrix} = \begin{bmatrix} \ldots 00000000000000010000001 \\ \ldots 11010111010100100000110 \\ \ldots 01000111011111100011011 0 \end{bmatrix} \tag{13}$$

$$= \begin{bmatrix} \ldots \underbrace{000000}_{0}\underbrace{000000}_{0}\underbrace{000010}_{2}\underbrace{000001}_{1} \\ \ldots \underbrace{110101}_{-11}\underbrace{110101}_{-11}\underbrace{001000}_{8}\underbrace{000110}_{6} \\ \ldots \underbrace{010001}_{17}\underbrace{110111}_{23}\underbrace{111000}_{-8}\underbrace{110110}_{-10} \end{bmatrix}$$

**Multiplication** Let's now examine matrix multiplication. Consider the product $P = A \times B^t$, where $B^t$ is the transpose of B. Thus the product becomes:

$$P = A \times B = \begin{bmatrix} 1 & 3 & 5 & 8 \\ 12 & 14 & 6 & 9 \\ 3 & 7 & 10 & 11 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 20 \\ 3 & 3 & 30 \\ 7 & 14 & 2 \\ 9 & 15 & 1 \end{bmatrix} \tag{14}$$

Again, using $AA$ and $BB$ to denote the compressed versions of the matrices, and $\times\times$ to denote the multiplication of bitstring matrices, the compressed product becomes:

$$P = AA \times BB = \begin{bmatrix} 4952 \\ 52841 \\ 14251 \end{bmatrix} \times\times \begin{bmatrix} 36073 & 36303 & 686145 \end{bmatrix} \tag{15}$$

$$P = AA \times BB = \begin{bmatrix} \ldots \underbrace{0001}_{1}\underbrace{0011}_{3}\underbrace{0101}_{5}\underbrace{1000}_{8} \\ \ldots \underbrace{1100}_{12}\underbrace{1110}_{14}\underbrace{0110}_{6}\underbrace{1001}_{9} \\ \ldots \underbrace{0011}_{3}\underbrace{0111}_{7}\underbrace{1010}_{10}\underbrace{1011}_{11} \end{bmatrix} \tag{16}$$

$$\times\times \begin{bmatrix} \ldots \underbrace{00001}_{1}\underbrace{00011}_{3}\underbrace{00111}_{7}\underbrace{01001}_{9} & \ldots \underbrace{00001}_{1}\underbrace{00011}_{3}\underbrace{01110}_{14}\underbrace{01111}_{15} & \ldots \underbrace{10100}_{20}\underbrace{11110}_{30}\underbrace{00010}_{2}\underbrace{00001}_{1} \end{bmatrix}$$

To calculate $P[1,1]$ which in uncompressed form is $1\times1+3\times3+5\times7+8\times9$, we first must extract the numbers from the bitstrings and then proceed with the linear combination. The extraction will make use of masks as described before. Let tempA and tempB store the values of blocks 1 and 3 of AA[1,1] and BB[1,1]. Now we need to determine the size of the blocks (bit-length) containing each element of the resulting matrix. As demonstrated before, in the worst case, the product will have a bit-length which is the sum of the bit-lengths of the operands. For this example this length is $4 + 5 = 9$, however, we also have three additions for each element,adding a total of 3 extra bits, thus we end up with a bit-length of 12 for each element of the product. Having determined the bit-length of the product, we can now we can do the actual calculations and store the results.

The linear combination of elements which generates $P[1,1]$ is detailed below. We use temporary variables $tempP_i$ to store the products before adding them, where $i$ is the product being calculated.

$$tempA = \ldots 0001|0011|0101|1000$$
$$tempB = \ldots 00001|00011|00111|01001$$

$$tempA = \ldots \mathbf{0001}|0011|0101|1000$$
$$tempB = \ldots \mathbf{00001}|00011|00111|01001$$
$$tempP_1 = \ldots 000000001$$

$$tempA = \ldots 0001|\mathbf{0011}|0101|1000$$
$$tempB = \ldots 00001|\mathbf{00011}|00111|01001$$
$$tempP_2 = \ldots 000001001$$

$$tempA = \ldots 0001|0011|\mathbf{0101}|1000$$
$$tempB = \ldots 00001|00011|\mathbf{00111}|01001$$
$$tempP_3 = \ldots 000100011$$

$$tempA = \ldots 0001|0011|0101|\mathbf{1000}$$
$$tempB = \ldots 00001|00011|00111|\mathbf{01001}$$
$$tempP_4 = \ldots 001001000$$

$$tempP1 = \ldots 0000000001$$
$$tempP2 = \ldots 0000001001$$
$$soma = \ldots 0000001010 \text{ (10 bits)}$$

$$tempP3 = \ldots 00000100011$$
$$soma = \ldots 00000001010$$
$$soma = \ldots 00000101101 \text{ (11 bits)}$$

$$
\begin{aligned}
tempP4 &= \ldots 000001001000 \\
soma &= \ldots 000000101101 \\
soma &= \ldots 000001110101 \text{ (12 bits)}
\end{aligned}
$$

So, $P[1,1]$ is 000001010101, and the entire matrix $P$ becomes:

$$
P = \begin{bmatrix} 1426882688 \\ 2970686121 \\ 3239350573 \end{bmatrix} = \begin{bmatrix} \ldots 00000101010100001100100000010000000 \\ \ldots 00001011000100010001000100101010100 1 \\ \ldots 00001100000100010100100100010010 1101 \end{bmatrix}
$$

(17)

$$
= \begin{bmatrix} \ldots \underbrace{000001010101}_{117} \underbrace{000011001000}_{200} \underbrace{000010000000}_{128} \\ \ldots \underbrace{000010110001}_{177} \underbrace{000100010001}_{273} \underbrace{001010101001}_{681} \\ \ldots \underbrace{000011000001}_{193} \underbrace{000101001001}_{329} \underbrace{000100101101}_{301} \end{bmatrix}
$$

Again we can see that P is already in compressed form, which confirms that the entire operation was conducted without decompressing the data.

The division operation with matrices will be described in a subsequent paper after we describe how to represent and operate with floating-point numbers.

In the examples given above, we only used the SM method, but the same procedure can be easily adapted to the VLB method.

Thus we complete our demonstration of how to perform basic arithmetic operations with bitstring compressed scalars and matrices.

# References


[1] Flores I (1963) The logic of computer arithmetic, volume 1063. Prentice-Hall.