Bachelor thesis
Computing Science

Radboud University

Bit-scale instead of byte-scale integer arrays in C

Author:
Michiel Kraan
s1023132

Supervisor/First assessor:
Professor, Sven-Bodo Scholz
sbscholz@gmail.com

Second assessor:
Assistant Professor, Peter Achten
p.achten@cs.ru.nl

July 18, 2023

Abstract

When working with integer arrays, sizes per element are typically limited to the size of a byte or multiple bytes, a byte being 8 bits in total. When using booleans, where only one bit is needed to signify `true` or `false`, a byte is typically stored, which means that 8 times as much memory is used as is actually needed.

This causes two problems. One is obvious, namely that more memory is used than is actually needed. The other is the fact that memory reading and writing is often multiple magnitudes slower than performing operations on the data, resulting in operation speed being heavily reliant on how much data can be fit into a given amount of memory.

This paper researches dense storage of integer arrays, both of booleans and larger integers such as of 2 bits, and tries various tricks to bundle reads from and writes to the same address in memory in order to save time. In theory, huge performance gains of up to 8 times should be possible as long as the operations are done efficiently, because up to 8 times as little memory is needed.

As it turns out, obtaining this performance is not as easy as expected, since in most scenarios more than just plain iteration over arrays needs to be done, in which case memory speed becomes less relevant, and operations need to be properly optimized to even achieve performance identical to a standard implementation.

However, by using clever techniques it definitely is possible to achieve performance close to that of a standard implementation, and in some specific scenarios even achieve a noticeable performance improvement. This shows that dense integer storage is a viable way to save memory without losing time, so it is worth exploring in areas where this may be relevant, such as in neural networks, where several papers have already found that using integers rather than floating point numbers does not come at any cost.

Contents

Chapter 1

Introduction

There is an ongoing hype in neural network-related research around using smaller floating-point numbers and even integers in network training, which is fuelled by the desire to reduce the storage size of neural networks, as well as to improve performance due to the improved data throughput because of smaller data. And not just in the field of neural networks, but in the broader space of computing, there has always been an interest in saving memory wherever possible, since storage can be expensive or even unavailable, and in the business world time equals money, so shrinking memory, which results in more throughout, can save a lot of money. This desire for a reduced memory footprint has inspired this paper to research using non-typical-sized integers in arrays, particularly in the C language, since they offer a way to reduce memory usage and produce code that can greatly outperform traditional arrays, because up to 8 times as little memory needs to be processed.

To elaborate, an array of booleans only needs a single bit per element, but due to hardware design, each boolean is stored as a byte, i.e. eight bits. For this paper, I create a series of C functions that can allocate and alter arrays that only use the amount of bits per element that the user needs, using data shifting and masking to store and retrieve values.

By doing this, the amount of memory used can be significantly reduced, and by using some clever mechanisms, certain operations can be executed in bulk, which has the potential to achieve performance similar to or even better than when using traditional arrays. This paper will discuss why and how such operations are implemented, and compare the performance between these functions and their traditional counterpart, to see how useful such an implementation is. In areas where memory is limited, this has the potential to greatly free up space, and if it could achieve better or similar performance, it can be appealing to a much larger audience.

Chapter 2

Notation

Throughout this document, I am using several notations for operations or number representation:

- x_2 where $x \in \{0,1\}^*$ means that the number x is not in decimal, but rather in binary notation. For instance, 101_2 means the number 5 $(1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1)$.
- $x \| y$ means x concatenated with y. For instance, $101_2 \| 010_2 = 101010_2$. Keep in mind that $0 = 0_2$ and $1 = 1_2$, so it can happen that $0 \| 1001_2$ is used, which is equal to $0_2 \| 1001_2$.
- x_2^n where $x \in \{0,1\}$ means the digit x repeated n times in binary. For instance, 1_2^5 means 11111_2 which is 31. Note that x is always a single digit, so something like $01_2^3 = 0111_2 = 7$. I avoid this concise notation though due to its confusing appearance, and instead use $0||1_2^3 = 0111_2$.
- $x \ll y$ and $x \gg y$ mimic the bitwise shifts in C. $x \ll y$ means x shifted y bits to the left, and $x \gg y$ means x shifted y bits to the right. Both these operations use zeroes to fill in new bits. For instance:

  $$3 \ll 2 = 11_2 \ll 2 = 1100_2 = 12$$

  $$34 \gg 3 = 100010_2 \gg 3 = 100_2 = 4$$

- $x \,\&\, y$ and $x \mid y$ represent bitwise AND and OR respectively. For instance:

  $$6 \,\&\, 12 = 0110_2 \,\&\, 1010_2 = 0010_2 = 2$$

  $$6 \mid 12 = 0110_2 \mid 1010_2 = 1110_2 = 14$$

- $\sim x$ is the bitwise NOT operation. For instance:

  $$\sim 6 = \sim 0110_2 = 1001_2 = 9$$

In several places, "little-endian" is mentioned, which specifies in which order the bytes (not bits) of integer types larger than a single byte are stored in memory by the computer. Little-endian means that the bytes are stored in ascending order from least significant bits (the bits that have the smallest value) to most significant bits (the bits that have the highest value). The bits within a byte itself are always ordered most significant bit leftmost, and least significant rightmost.

For instance, the 32-bit integer 00000000 00001111 11110000 11111111 is stored as follows in little-endian: address $a = 11111111$, $a+1 = 11110000$, $a+2 = 00001111$, $a+3 = 00000000$. In contrast, in big-endian, the same number is stored as follows: $a = 00000000$, $a+1 = 00001111$, $a+2 = 11110000$, $a+3 = 11111111$.

Chapter 3

The Traditional Way

In C, C++, Rust, Java, Python, and most other languages, the smallest usable unit is the byte, which is 8 bits in total. This has been the case for a long time, because most if not all common computers, from desktops to smartphones, read from and write to memory using addresses that each store exactly one byte, no more, no less. Making it possible to address individual bits directly can increase the complexity and power consumption of hardware, so it was decided that memory is only accessible on a per-byte basis.

If fewer than 8 bits are needed per element, a byte is used. In many scenarios, this is no issue, since often only a handful of those elements are used and the time or storage saved by using fewer bits is negligible in such scenarios. However, when working with large arrays of such data, the amount of redundant memory becomes much more significant. Take booleans for instance. These are either `true` or `false`, which are represented as 1 and 0 in binary. This in theory only needs 1 bit, but in reality booleans are usually stored as a byte, which is 8 bits. 7 out of 8 of these bits, or 87.5%, is redundant! On top of that, given that memory is relatively slow compared to how fast processors work, the speed of computations can be significantly reduced by having such large amounts of extra data.

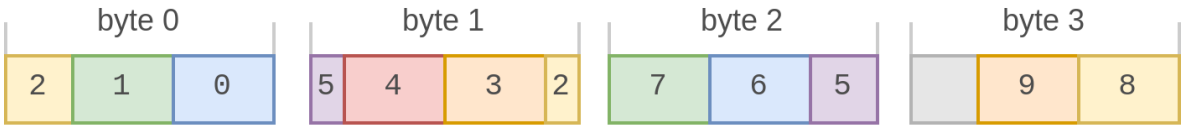
Quite clearly, it is worth investigating just how fast a compact storage method can become, to see if if it's feasible to use. In theory it should be possible to achieve performance up to 8 times as fast, but it will depend on the way the data is processed.

Chapter 4

Compact Storage

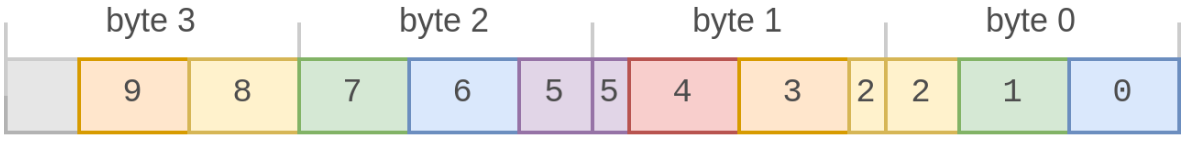
4.1 Memory layout

We first need to define the way in which numbers are stored in memory. Values within a byte are stored in ascending order from least significant bit to most significant bit, where bytes with lower indices contain the first values of the array, and higher-index bytes contain values later in the array. For instance, suppose each number requires three bits, and we have an array of 10 elements. This would look like the following:



Where numbers 0 through 9 are the indices of the values stored in the array.

There is reason why having bytes in this order is useful, apart from it being the most straightforward. Most modern computers store numbers in memory in little-endian form. If an integer type of more than a single byte is used in code, these bytes are aligned so that the lowest address contains the least significant bits. Suppose we interpreted this array as a 32-bit integer (`uint32_t`), it would become the following integer:



As can be seen, numbers that were previously spread across bytes, are now correctly aligned. Because of this, obtaining and updating values in arrays with more than 8 bits per number can be done by treating arrays as 16, 32, or 64 bit integer arrays, instead of having to construct and destruct values from or to individual bytes, which can greatly improve performance.

The vast majority of computers today are little-endian, and so the focus of this thesis is on that memory layout. The code will not work for big-endian computers. For those computers, an option could be to handle bytes in reverse, so that bytes containing the lower-index values

are at a higher address, and the end of the vector is at a lower address. But that is more complex, and useful in very few to no scenarios.

4.2 Allocating arrays

In order to find the number of bytes required to represent an array, we need to know the bits required per number, and the total number of elements in the array.

For the sake of simplicity, the following definitions are for arrays with $bpn \leq 8$, and reads and writes from and to a_x produce and require 8 bit integers. How the same can be done for $bpn \leq 64$, along with accompanying definitions, will be discussed in section 4.5.

The total number of bits required is calculated as follows:

$$b = bpn \cdot \prod_{d=1}^{D} n_d \tag{4.1}$$

Where b is the total number of bits, bpn is the amount of bits per number, D is the dimensionality (amount of dimensions) of the array, and n_i is the size of dimension i of the array.

The C language (and most, if not all, other programming languages) can only represent arrays using bytes, so we need to transform this bit count into bytes in order to allocate arrays. To obtain the fewest amount of bytes required, the number of bits needs to divided by 8 (fractional, not integer division, so we obtain a fractional result), and rounded up:

$$B = \lceil b \div 8 \rceil$$

However, as in section 4.1 we discussed being able to treat the data as arrays of not just 8, but also 16, 32, and 64 bit integers, in order to improve performance. When reading or writing at the end of the array, this can cause problems, since using the above calculation of b, the number of bits allocated may not be a multiple of one of the above integer sizes, and thus reading and/or writing at the end of the array may go out of bounds.

To prevent this, there are two options:

- Treat the entire array as 64-bit integers, and the last few bytes as 8-bit integers. This ensures that the minimum amount of memory is actually used.
- Allocate slightly more bytes so that B is a multiple of 8, which makes it possible to treat the entire array as one consisting of 64-bit integers without reading or writing out of bounds.

The problem with the former option is that it requires significantly more code while only saving up to 7 bytes. So, I decided to always allocate a multiple of 8 bytes, to benefit from the reduced code at the cost of a small amount of redundant data.

The calculation of B then becomes:

$$B = \lceil b \div 64 \rceil \cdot 8 \tag{4.2}$$

As an example, suppose we want a two-dimensional array of 3-bit integers, with twenty rows and ten columns. The number of bits required is calculated as follows:

$$b = 3 \cdot \prod_{d=1}^{2} n_d = 3 \cdot 20 \cdot 10 = 600$$

The number of bytes required then becomes:

$$B = \lceil 600 \div 64 \rceil \cdot 8 = \lceil 9.375 \rceil \cdot 8 = 10 \cdot 8 = 80$$

In comparison, using an entire byte per number would require $20 \cdot 10 = 200$ bytes, which means that this compact storage's memory footprint is only 40% of a regular array's.

4.3 Obtaining values

We take the example from before to determine how to obtain individual values, and assign some random values:

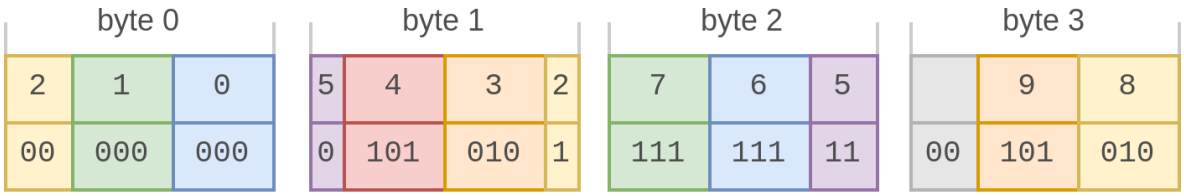


Figure 4.1: Array memory layout

Suppose we want to obtain the value at position 3 in this array, whose value is, for the purpose of this example, conveniently stored entirely in a single byte. We make the following definitions:

- a is the array
- a_i is the byte at position i in a. So there are 4 values for a_i: a_0 up to and including a_3.
- v_j^a is the value in a at position j. This is the actual 3-bit value, not the raw byte at the given position. So there are a total of 10 values for v_j^a: v_0^a up to and including v_9^a.

To obtain v_3^a, we need to first determine two things: which byte the element is located in (i in a_i), and which bit the element starts at.

Luckily, this can be done using simple calculations. We first multiply the index by the number of bits per element to obtain the absolute bit position:

$$p_i = i \cdot bpn \tag{4.3}$$

To split this into a byte and bit index, we have two definitions:

$$p_i^B = p_i \gg 3 \ (= \lfloor p_i / 8 \rfloor) \tag{4.4}$$

$$p_i^b = p_i \ \& \ 111_2 \ (= p_i \bmod 8) \tag{4.5}$$

Where p_i^B is the byte index, and p_i^b is the bit index (within the byte).

Back to the example, for position 3 this gives:

$$\begin{aligned} p_3 &= 3 \cdot 3 = 9 \\ p_3^B &= 9 \gg 3 = 1001_2 \gg 3 = 1 \\ p_3^b &= 9 \ \& \ 111_2 = 1001_2 \ \& \ 111_2 = 0001_2 = 1 \end{aligned}$$

This means that v_3^a is located in byte 1, starting at bit 1, which can be visually confirmed in figure 4.1. (v_2^a takes up one bit in byte 1, so the value v_3^a, that comes right after, starts at bit 1.)

Next, we need to shift the value of byte 1 to the right, so that a_3 starts at bit 0, and then the final result needs to be masked so that the bits not belonging to a_3 (in this case, a_4 and a_5) are set to zero:

$$v_i^a = (a_{p_i^B} \gg p_i^b) \ \& \ m^a$$

Where $a_{p_i^B}$ is the byte in array a at position p_i^B and m^a is a mask which sets all bits above bpn to 0, which can be calculated as follows:

$$m^a = 1_2^{bpn_a} \tag{4.6}$$

In the case of 3-bit values, this becomes $1_2^3 = 111_2$.

In the example from before, suppose $a_1 = 01010101_2$. I use underlines here to indicate which parts remain after the operation that they're involved in, for clarity.

$$v_3^a = (\underline{01010101}_2 \gg 1) \ \& \ 111_2 = 0101\underline{010}_2 \ \& \ 111_2 = 010_2 = 2$$

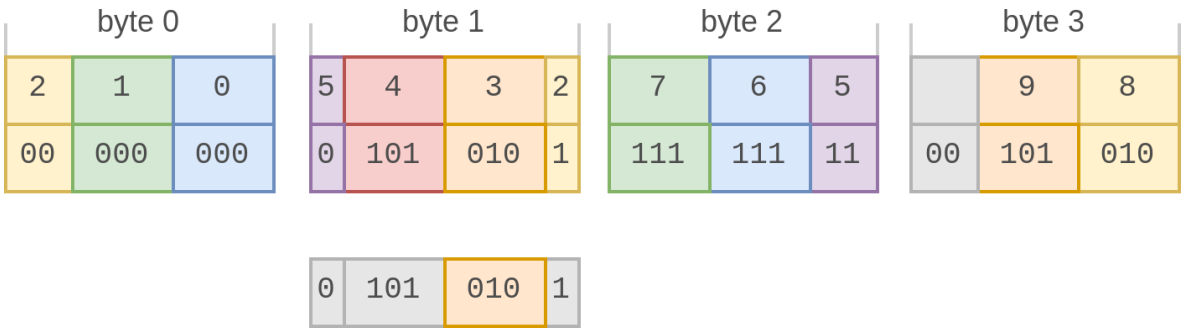


Figure 4.2: Location of v_3^a

Now suppose that we want to obtain v_5^a. For this, we still obtain the byte and bit in the same way:

$$\begin{aligned} p_5 &= 5 \cdot 3 = 15 \\ p_5^B &= 15 \gg 3 = 1111_2 \gg 3 = 1 \\ p_5^b &= 15 \ \& \ 111_2 = 1111_2 \ \& \ 111_2 = 0111_2 = 7 \end{aligned}$$

However, we can see in figure 4.1 that the value is now spread across two bytes. To determine whether this is the case programmatically, we use the following condition:

$$p_i^b + bpn > 8 \tag{4.7}$$

If this holds, we need to combine the lower bits (in a_i) with the higher bits (in a_{i+1}) of this number. The shifting of the lower byte is still performed like before, but this needs to be combined with the data in a_{i+1}. We do this by shifting the higher byte to the left by $8 - p_i^b$ bits, because this is the number of bits that the value occupies in a_i; after this number of bits the part stored in a_{i+1} starts. Finally, we can mask in the same way to obtain the following formula:

$$v_i^a = ((a_{p_i^B} \gg p_i^b) \mid (a_{p_i^B+1} \ll (8 - p_i^b))) \;\&\; m^a$$

Suppose $a_1 = 01010101_2$ like before, and $a_2 = 11111111_2$. Then (again, underlines for clarity):

$$\begin{aligned}
v_5^a &= ((\underline{0}1010101_2 \gg 7) \mid (11111111_2 \ll (8-7))) \;\&\; 111_2 \\
&= (00000000_2 \mid (11111111_2 \ll \underline{(8-7)})) \;\&\; 111_2 \\
&= (00000000_2 \mid (1\underline{1111111}_2 \ll 1)) \;\&\; 111_2 \\
&= (00000000_2 \mid \underline{11111110}_2) \;\&\; 111_2 \\
&= 11111\underline{110}_2 \;\&\; 111_2 \\
&= 110_2
\end{aligned}$$

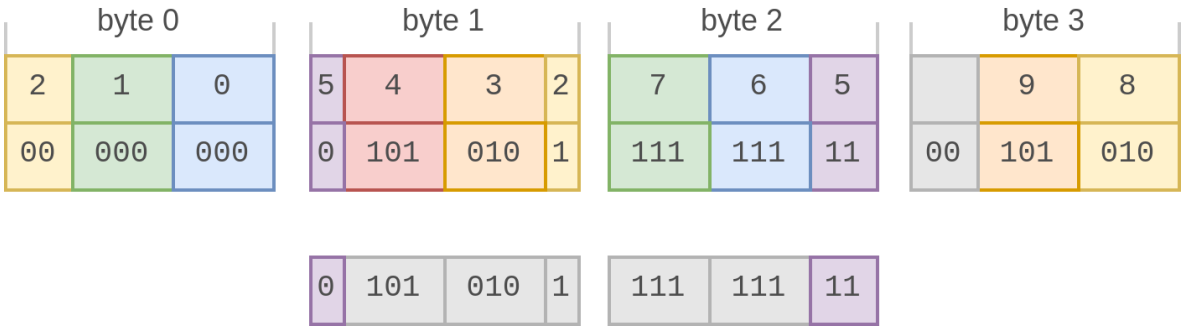


Figure 4.3: Location of v_5^a

To make a general definition for v_i^a, we combine the two cases above to arrive at:

$$v_i^a = \begin{cases} (a_{p_i^B} \gg p_i^b) \;\&\; m^a & \text{if } p_i^b + bpn \leq 8 \\ ((a_{p_i^B} \gg p_i^b) \mid (a_{p_i^B+1} \ll (8 - p_i^b))) \;\&\; m^a & \text{otherwise} \end{cases} \tag{4.8}$$

4.4 Setting values

Setting values works similarly to obtaining values. We define i at the to-be-updated position, and v as the new value. First, we obtain p_i^B and p_i^b in the same way as before.

We then need to perform two steps to update $a_{p_3^B}$:

1. Mask out the existing byte $a_{p_3^B}$, so that bits used by the value are set properly, but other bits remain unchanged. We can generate this mask as follows:
   (a) Take m^a;
   (b) Shift to p_i^b. This makes it so all bits occupied by v_i^a are 1, and the rest are 0;
   (c) Invert, so that 1s and 0s are swapped.

   Then this mask can be AND-ed with the byte currently in the array:

$$a_{p_i^B} \;\&\; (\sim (m^a \ll p_i^b))$$

2. Mask and shift the new value to p_i^b. We can do this by AND-ing the value with the mask, and then shifting to the left by p_b:

$$(v \;\&\; m^a) \ll p_i^b$$

Finally, we combine these using an OR to obtain:

$$a_{p_i^B} := (a_{p_i^B} \;\&\; (\sim (m^a \ll p_i^b))) \mid ((v \;\&\; m^a) \ll p_i^b) \tag{4.9}$$

In the example from before, suppose we want to set v_3^a to 5 ($= 101_2$):

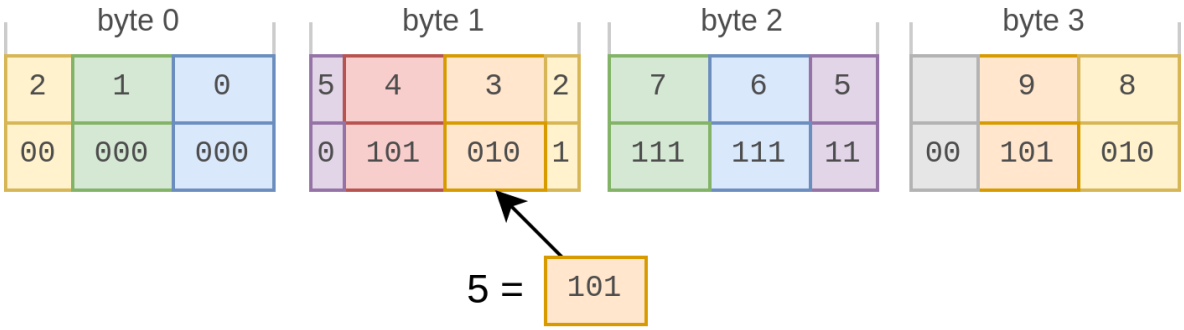


Figure 4.4: Assigning to v_3^a

With $a_1 = 01010101$, we get:

$$\begin{aligned}
a_{p_3^B} &:= (a_{p_3^B} \;\&\; (\sim (m^a \ll p_3^b))) \mid ((v \;\&\; m^a) \ll p_3^b) \\
&:= (01010101 \;\&\; (\sim (\underline{111}_2 \ll 1))) \mid ((101_2 \;\&\; 111_2) \ll 1) \\
&:= (01010101 \;\&\; (\sim 0000\underline{1110}_2)) \mid (\underline{101}_2 \ll 1) \\
&:= (0101\underline{0101} \;\&\; 1111\underline{0001}_2) \mid \underline{1010}_2 \\
&:= 0101\underline{0001} \mid \underline{1010}_2 \\
&:= 0101\underline{1011}
\end{aligned}$$

In some cases, we need to update $a_{p_3^B+1}$ as well. Condition 4.7 is used again to see whether the value is spread across bytes. If this holds, the update of $a_{p_3^B}$ needs no changing, but $a_{p_3^B+1}$ must now also be updated. This consists of two steps:

1. Mask out the existing byte, which sets the bits occupied by v_i^a to zero:

$$a_{p_i^B+1} \;\&\; (\sim (m^a \gg (8 - p_i^b)))$$

2. Shift the new value to the correct position:

$$(v \;\&\; m^a) \gg (8 - p_i^b)$$

Now combine these to obtain:

$$a_{p_i^B+1} := (a_{p_i^B+1} \;\&\; (\sim (m^a \gg (8 - p_i^b)))) \mid ((v \;\&\; m^a) \gg (8 - p_i^b)) \tag{4.10}$$

In the example, suppose we want to set the value at position 5 to 1 ($= 001_2$):

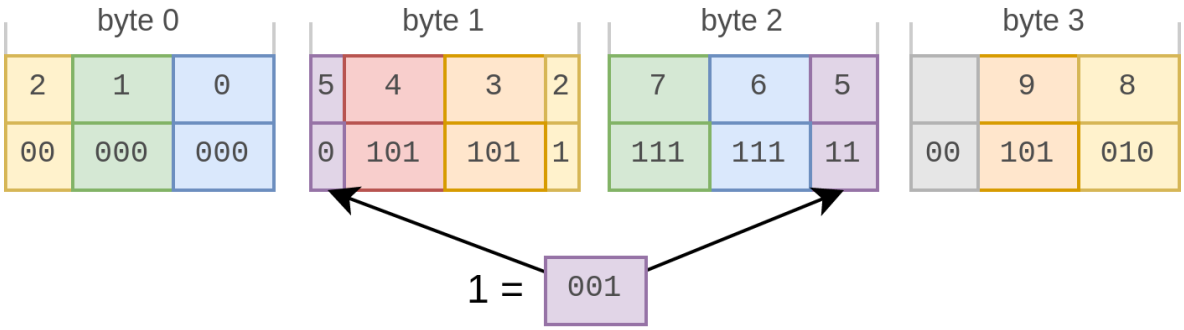


Figure 4.5: Assigning to v_5^a

We get:

$$\begin{aligned}
a_{p_5^B} &:= (a_{p_5^B} \;\&\; (\sim (m^a \ll p_5^b))) \mid ((v \;\&\; m^a) \ll p_5^b) \\
&:= (01011011_2 \;\&\; (\sim (\underline{111}_2 \ll 7))) \mid ((\underline{001}_2 \;\&\; \underline{111}_2) \ll 7) \\
&:= (01011011_2 \;\&\; (\sim \underline{1}0000000_2)) \mid (\underline{001}_2 \ll 7) \\
&:= (\underline{0}1011011_2 \;\&\; \underline{0}1111111_2) \mid \underline{1}0000000_2 \\
&:= \underline{0}1011011_2 \mid \underline{1}0000000_2 \\
&:= \underline{1}1011011_2
\end{aligned}$$

$$\begin{aligned}
a_{p_5^B+1} &:= (a_{p_5^B+1} \;\&\; (\sim (m^a \gg (8 - p_5^b)))) \mid ((v \;\&\; m^a) \gg (8 - p_5^b)) \\
&:= (11111111_2 \;\&\; (\sim (\underline{111}_2 \gg (8 - 7)))) \mid ((\underline{001}_2 \;\&\; \underline{111}_2) \gg (8 - 7)) \\
&:= (11111111_2 \;\&\; (\sim 0000001\underline{1}_2)) \mid (\underline{001}_2 \gg (8 - 7)) \\
&:= (1111111\underline{1}_2 \;\&\; 11111\underline{00}_2) \mid \underline{00}_2 \\
&:= 11111\underline{00}_2 \mid \underline{00}_2 \\
&:= 11111\underline{00}_2
\end{aligned}$$

4.5 Values using more than 8 bits

The functions explained in the previous sections are for arrays with $bpn \leq 8$. They can be generalized to arrays with elements of up to 16, 32 or 64 bits as follows.

For the following combinations:

- $(bits, shift^B, mask^b) = (8, 3, 111_2)$
- $(bits, shift^B, mask^b) = (16, 4, 1111_2)$
- $(bits, shift^B, mask^b) = (32, 5, 11111_2)$
- $(bits, shift^B, mask^b) = (64, 6, 111111_2)$

We define:

$$p_i^B = p_i \gg shift^B \ (= \lfloor p_i / bits \rfloor) \tag{4.11}$$

$$p_i^b = p_i \ \& \ mask^b \ (= p_i \bmod bits) \tag{4.12}$$

a_x now treats array a as an array of $bits$ bits, so reading produces an integer of 8, 16, 32 or 64 bits rather than just 8, and the same goes for writing.

Now, to read a value:

$$v_i^a = \begin{cases} (a_{p_i^B} \gg p_i^b) \ \& \ m^a & \text{if } p_i^b + bpn_a \leq \underline{bits} \\ ((a_{p_i^B} \gg p_i^b) \mid (a_{p_i^B+1} \ll (\underline{bits} - p_i^b))) \ \& \ m^a & \text{otherwise} \end{cases} \tag{4.13}$$

And to write a value:

$$a_{p_i^B+1} := (a_{p_i^B+1} \ \& \ (\sim (m^a \gg (\underline{bits} - p_i^b)))) \mid ((v \ \& \ m^a) \gg (\underline{bits} - p_i^b)) \tag{4.14}$$

Chapter 5

Batch Operations

When working with large datasets, it is very common to perform similar or identical operations on (parts of) the entire dataset at once. Rather than doing so on a per-element basis, with some clever techniques it's possible to bundle multiple reads or writes into fewer operations, which depending on the task at hand could improve performance by quite a lot. Therefore, the project also defines several functions for manipulating and reading (parts of) an array.

5.1 Filling with a single value

The first of these functions fills a (sub)array with a single constant value c. This is done efficiently in repeated 64-bit steps, however for simplicity it will be explained using 8-bit steps. We define the function:

$$bulk\_fill(a, i, j, c) \tag{5.1}$$

where a is the target array, i is the starting position (inclusive), j is the ending position (exclusive), and c is the value to insert.

First, an 8-bit number f_0 is generated, which is repeatedly filled with c from LSB (Least Significant Bit = the rightmost) to MSB (Most Significant Bit = the leftmost). This produces an integer where all values in byte 0 of the array are c. Suppose we want to fill our example array (with $bpn = 3$) in range $[4, 9)$ with value $6 = 110_2$. We get:

$$f_0 = 10110110_2 \ (= (\underline{\mathbf{10}}\|\underline{\mathbf{110}}\|\underline{\mathbf{110}})_2)$$

Which can be seen in byte 0 in the following image:

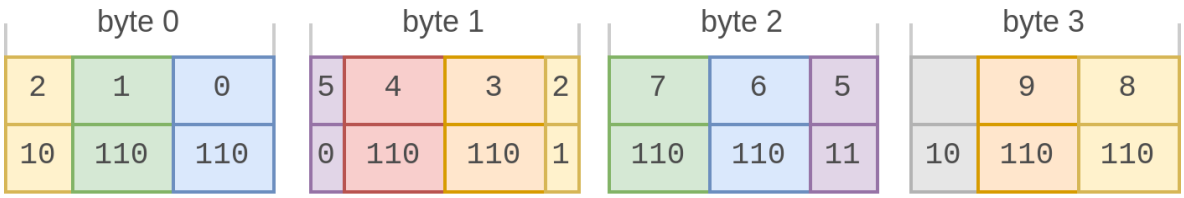


Figure 5.1: Array filled with 6

We also store two integers:

- h is the value of p_i^b for i the highest index for which $p_i^B = 0$ ($i = 2$, $p_i^b = 6$ in this case). We can calculate this using:

$$h = \lfloor 7/bpn \rfloor \cdot \ bpn \ (= 7 - (7 \text{ MOD } bpn) \tag{5.2}$$

  The reason 7 is used instead of 8, is because when bpn divides 8, 8 MOD $bpn = 0$, and so the highest index returned is 8, but that is already part of the next byte. By using 7, we obtain the value that comes at the end of f_0, and is still (partially) in f_0.

- s the number of bits that this highest-located value is missing in a_0:

$$s = h + bpn - 8 \tag{5.3}$$

  which is $9 - 8 = 1$ in this case.

To see what this means, suppose we want to know f_1. Since in a_0 the highest value is 10_2 (and thus it is missing a 1, which is stored in a_1), the missing 1 will be stored at position 0 in a_1, and the value after that starts at position 1 in a_1 (highlighted in bold below):

$$f_1 = 01101101_2 \ (= (0 \| 110 \| \mathbf{110} \| 1)_2)$$

The highest located value v_j^a in f_1 is now missing 2 bits (there's only 0 in f_1), which means that in f_2 the value v_{j+1}^a is stored at bit position 2 (highlighted in bold):

$$f_2 = 11011010_2 \ (= (110 \| \mathbf{110} \| 10)_2)$$

In f_3, this would logically become 3, however since each value is only 3 bits, the lowest value in there is stored between $[0, 3)$, so the lowest located value is at bit 0, which is 3 MOD 3:

$$f_3 = 10110110_2 \ (= (10 \| 110 \| \mathbf{110})_2)$$

This s value therefore denotes the shift in offset from f_n to f_{n+1}.

If we want to calculate f_n for a given p_i^b, we determine the position of the value with the lowest p_b that starts within the current byte:

$$o_{p_i^B} = p_i^b \text{ MOD } bpn \tag{5.4}$$

In the example, our starting index is 4, which produces:

$$\begin{aligned} p^4 &= 4 \cdot 3 = 12 \\ p_4^B &= 12 \gg 3 = 1 \\ p_4^B &= 12 \ \& \ 7 = 4 \\ o_{p_4^B} &= 4 \text{ MOD } 3 = 1 \text{ (which is } p_3^B) \end{aligned}$$

To obtain f_n, we need to generate a value like f_0, however this time rotated (not shifted) to position o_n. To obtain the bits left of o_n, we left-shift f_0 by o_n. To obtain the bits right of o_n, we right-shift f_0 by $(h - o_n)$. This makes:

$$f_n = (f_0 \ll o_n) \mid (f_0 \gg (h - o_n)) \tag{5.5}$$

In the example, this gives:

$$\begin{aligned} f_1 &= (10110110_2 \ll 1) \mid (10110110_2 \gg (6-1)) \\ &= 01101100_2 \mid 00000101_2 \\ &= 01101101_2 \end{aligned}$$

This indeed matches f_1 from figure 5.1!

To calculate o_{n+1}, we add s modulo bpn to o_n:

$$o_{n+1} = (o_n + s) \text{ MOD } bpn \tag{5.6}$$

Since $s < bpn$ (because the shift in each 64-bit step is always smaller than the size of the elements) and $o_n < bpn$ (because of the MOD requirement), the result of the addition in o_{n+1} above will always be $< (2 \cdot bpn)$, and so this MOD operation can be simplified with a check for $\geq bpn$ and a single subtraction. This is important for performance, since the MOD operation is computationally expensive, and a single comparison and subtraction are not.

We distinguish between two cases.

If $p_i^B < p_j^B$, $a_{p_i^B}$ and $a_{p_j^B}$ need to only partially be modified, and retain the unchanged values:

$$a_q \quad := \quad \begin{cases} (f_q \ \& \ (1_2^8 \ll p_i^b)) \mid (a_q \ \& \ (1_2^8 \gg (8 - p_i^b)) & \text{for } q = p_i^B \\ f_q & \text{for } p_i^B < q < p_j^B \\ (a_q \ \& \ (1_2^8 \ll p_j^b)) \mid (f_q \ \& \ (1_2^8 \gg (8 - p_j^b)) & \text{for } q = p_j^B \end{cases} \tag{5.7}$$

Otherwise (if $p_i^B = p_j^B$), both values before and after the range need to be retained:

$$a_{p_i^B} := (f_{p_i^B} \ \& \ m') \mid (a_{p_i^B} \ \& \ (\sim m')) \quad \text{where } m' = (1_2^8 \ll p_i^b) \ \& \ (1_2^8 \gg (8 - p_j^B)) \tag{5.8}$$

In the example, suppose $(a_1, a_2, a_3) = (ABCDEFGH_2, IJKLMNOP_2, QRSTUVWX_2)$:

$$\begin{aligned} a_1 &:= (01101101_2 \ \& \ 11110000_2) \mid (ABCDEFGH_2 \ \& \ 00001111_2) \\ &= 01100000_2 \mid 0000EFGH_2 \\ &= 0110EFGH_2 \\ a_2 &:= 11011011_2 \\ a_3 &:= (QRSTUVWX_2 \ \& \ 11111000_2) \mid (10110110_2 \ \& \ 00000111_2) \\ &= QRSTU000_2 \mid 00000110_2 \\ &= QRSTU110_2 \end{aligned}$$

If we concatenate these numbers in little-endian order, it can be seen that all values have been successfully set:

$$a_3 \| a_2 \| a_1 = QR\ STU\ 110 \| 110\ 110\ 11 \| 0\ 110\ EFG\ H_2$$

5.2 Filling with different values

If an array should be filled with various values instead of a single value, the method of the previous section cannot be used. Write operations are costly though, so ideally there would be as few of these as possible when filling a larger (part of an) array. What we can do, is rather than setting individual elements, we update $a_{p_i^B}$ for any values (partially) stored in byte p_i^B. This ensures that each memory address is only written to once, and should provide optimal memory performance.

We define a function which populates a (sub)array with varying values:

$$bulk\_set(a, i, j, g)$$

where a is the target array, i is the starting position (inclusive), j is the ending position (exclusive), and g is a function which takes a position as input and returns the value to put in that position:

$$g(i) :: unsigned\ int$$

When filling an array in the range $[i, j)$, if $p_i^B < p_j^B$, we need to:

- In the first byte, keep the bits $< p_i^b$, and update the rest
- In the intermediate bytes, we can update all bits
- In the last byte, keep the bits $\geq p_j^B$, and update the rest

We define the following updates:

$$a_{p_i^B} := (a_{p_i^B}\ \&\ \sim(1_2^8 \ll p_i^b)) \mid (\sum_{\substack{k \in K \\ \text{where } K = \{k \mid p_k^B = p_i^B \wedge p_k^b \geq p_i^b\}}} (g(k)\ \&\ m^a) \ll p_k^B) \tag{5.9}$$

$$\underset{\text{for } q \in (p_i^B, p_j^B)}{a_q} := (\sum_{\substack{k \in K \\ \text{where } K = \{k \mid p_k^B = q\}}} (g(k)\ \&\ m^a) \ll p_k^B) \mid r_q \tag{5.10}$$

$$a_{p_j^B} := (a_{p_j^B}\ \&\ 1_2^8 \ll p_j^B) \mid (\sum_{\substack{k \in K \\ \text{where } K = \{k \mid p_k^B = p_j^B \wedge p_k^b < p_j^b\}}} (g(k)\ \&\ m^a) \ll p_k^B) \mid r_{p_j^B} \tag{5.11}$$

r_q above is defined as:

$$r_q = \begin{cases} (f(k)\ \&\ m) \gg (8 - p_k^B) & \text{if } \exists k : p_k^B = q - 1 \wedge p_k^B + bpn > 8 \\ 0 & \text{otherwise} \end{cases} \tag{5.12}$$

Which is a possible value that starts in byte $p_k^B - 1$ and is partially stored in p_k^B. This needs to be manually added because its origin byte $p_k^B - 1$ is not p_k^B, and it needs to be shifted right rather than left.

In the case that $p_i^B = p_j^B$, where both higher and lower bits in the byte need to be retained, a different update is done:

$$a_{p_i^B} := (a_{p_i^B}\ \&\ (1_2^8 \ll p_j^B)\ \&\ \sim(1_2^8 \ll p_i^b)) \mid (\sum_{\substack{k \in K \\ \text{where } K = \{k \mid p_k^B = p_i^B\}}} (f(k)\ \&\ m) \ll p_k^B) \tag{5.13}$$

5.3 Computing values from two inputs

This function takes two input arrays a and b, and in a given range, computes an output per element using a given function, and stores it into output array c at a given offset.

This is something that works for a few specific operations, but it is also capable of treating the entire array as 64-bit integers, rather than individual elements. This means that certain operations (particularly bitwise and addition) can be done much quicker for datasets with small integer types, however certain other operations (such as multiplication) are either impossible or are too complex to be more efficient than element-wise steps.

Suppose we have two input boolean (1-bit) arrays and an output boolean array. Bitwise operations can be performed by simply performing the tasks on the 64-bit integers, the outcome of one bitwise operation does not depend on another bit.

It is also possible to perform addition, albeit with some extra steps:

- First, the two 64-bit integers are added together.
- In order to prevent a resulting value from an addition from overflowing into the next element, an additional redundant bit is needed per element. To see this in action, see the following example:

  Suppose we have arrays of two elements, both requiring 3 bits. Without a redundant bit, we could have the following scenario:

  $$00_2 \| 100_2 \| 100_2 + 00_2 \| 100_2 \| 100_2 = 01_2 \| 001_2 \| 000_2$$

  As can be seen, the additions done here result in a bit being carried to the next element, which produces unexpected results. By adding a redundant bit to each element, we would have the following:

  $$0100_2 \| 0100_2 + 0100_2 \| 0100_2 = 1000_2 \| 1000_2$$

  If we then set these extra bits back to 0 by AND-ing it with a mask $0111_2 \| 0111_2$, the redundant bit is restored back to 0:

  $$1000_2 \| 1000_2 \ \& \ 0111_2 \| 0111_2 = 0000_2 \| 0000_2$$

  It holds that for all additions between two numbers of b bits, at most $b + 1$ bits are needed to store the output. We can prove this as follows, by adding the two highest possible values that can be represented with 3 bits:

  $$\begin{aligned} 111_2 + 111_2 &< 1111_2 \\ (1000_2 - 1) + (1000_2 - 1) &< 1111_2 \\ 10000_2 - 10_2 &< 1111_2 \\ 1110_2 &< 1111_2 \end{aligned}$$

Which can be generalized to the following:

$$\begin{aligned} 1_2^b + 1_2^b &< 1_2^{b+1} \\ ((1 \| 0_2^b) - 1) + ((1 \| 0_2^b) - 1) &< 1_2^{b+1} \\ (1 \| 0_2^{b+1}) - 10_2 &< 1_2^{b+1} \\ (1_2^b \| 0) &< 1_2^{b+1} \end{aligned}$$

Since (1_2^{b+1}) requires $b+1$ bits, and the result of the addition is smaller than this value, the result requires $b+1$ bits. And since adding any other two lower values produces a lower result, the result of any addition between numbers of b bits requires at most $b+1$ bits.

This extra bit needs to be included when allocating an array, and the proper mask should be remembered for operations that will be performed on the array, so that the redundant bit will always remain 0.

- To make sure numbers spread across bytes are properly added, the carry obtained from addition needs to be tracked and passed to the next byte. With an initial $c = 0$, this can be obtained using the following:

$$ca_{s+1} := \begin{cases} 1 & \text{if } (v_s^a > 1_2^8 - v_s^b) \vee (ca_s = 1 \wedge v_s^a + v_s^b = 1_2^8) \\ 0 & \text{otherwise} \end{cases} \tag{5.14}$$

The function has the following parameters: input arrays a and b, output c, offsets i^a, i^b and i^c, count n, and function g:

$$g(v_s^a,\ v_s^b,\ m_s,\ ca_s) \to (v_s^c,\ ca_{s+1})$$

where:

- v_s^a and v_s^b are the input values from a and b, relative to i^a and i^b.
- v_s^m is the number mask at byte offset s. For arrays with no redundant bits per element (needed for addition), this is simply all 1s. For arrays with redundant bits, this is a mask where element bits are 1s, and redundant bits are 0s, correctly aligned to the values in v_s^a and v_s^b.
- ca is the input carry,
- v_s^c is the computed value, to be stored in c relative to i^c.
- ca_{s+1} is the new carry.

We want make it to when we write to c, we can do so in blocks of 64 bit integers, starting at bit position 0 within an array index, so that no extra masking needs to be done.

To accomplish this, we need to first process fewer than 64 bits so that the next bit index of c is at 0. The first step processes $64 - p_{i^c}^B$ bits, so that the next step starts at bit 0. From then on, values can be written to c in repeated 64-bit steps.

We define o_{step}^{array}, which denotes the starting bit index of an array for a given step. For c, this is simple:

$$o_s^c = \begin{cases} p_{i^c}^B & \text{if } s = 0 \\ 0 & \text{if } s > 0 \end{cases} \tag{5.15}$$

For a and b, the bit indices in the first step are also the starting bits of the operation. After that, given the fact that in the first step we process $64 - o_0^c$ bits, subsequent steps are this starting offset plus the number of bits processed, modulo 64.

$$o_s^a = \begin{cases} p_{i^a}^B & \text{if } s = 0 \\ (o_0^a + (64 - o_0^c)) \bmod 64 & \text{if } s > 0 \end{cases} \tag{5.16}$$

$$o_s^b = \begin{cases} p_{i^b}^B & \text{if } s = 0 \\ (o_0^b + (64 - o_0^c)) \bmod 64 & \text{if } s > 0 \end{cases} \tag{5.17}$$

We define the array indices for these updates accordingly:

$$B_a^s = \begin{cases} p_{i^a}^B & \text{if } s = 0 \\ B_a^0 + (1 \text{ if } (o_0^a \geq o_0^c) \text{ else } 0) & \text{if } s = 1 \\ B_a^1 + s - 1 & \text{if } s > 1 \end{cases} \tag{5.18}$$

$$B_b^s = \begin{cases} p_{i^b}^B & \text{if } s = 0 \\ B_b^0 + (1 \text{ if } (o_0^b \geq o_0^c) \text{ else } 0) & \text{if } s = 1 \\ B_b^1 + s - 1 & \text{if } s > 1 \end{cases} \tag{5.19}$$

$$B_c^s = p_{i^c}^B + s \tag{5.20}$$

We need to obtain the mask for each array index as well, which is done using the same method as when filling an array with a single value (although in this example it's in 64 bits rather than 8). We create adjusted versions of 5.2 and 5.3:

$$high = \lfloor 63/bpn \rfloor \cdot \ bpn \tag{5.21}$$

$$shift = high + bpn - 64 \tag{5.22}$$

Note the use of 63 rather than 64, for the same reason as 7 was used instead of 8.

The value passed to g in the first step repeated sequences of m starting at o_0^c. To obtain the correct mask at step s, we start with offset o_0^c, and after that use the $shift$ value to adjust this offset.

We first define the bit index of the lowest value in the array index of each step:

$$l_s = \begin{cases} o_0^c \text{ MOD } bpn & \text{if } s = 0 \\ (l_{s-1} + shift) \text{ MOD } bpn & \text{if } s > 0 \end{cases} \tag{5.23}$$

Then, we can define the initial and subsequent masks:

$$m_s = \begin{cases} \sum_{i=0}^{\lfloor 63/bpn \rfloor} (m \ll (i \cdot bpn)) & \text{if } s = 0 \\ (m_0 \ll l_s) \mid (m_0 \gg (high - l_s)) & \text{if } s > 0 \end{cases} \tag{5.24}$$

Finally, we can define the actual updates.

When $B_0^c < p^B_{i^c+n}$:

$$(c_{B_s^c}, ca_{s+1}) := \begin{cases} ((c_{B_s^c} \mathbin{\&} \sim m(o_0^c)) \mid (x \mathbin{\&} m(o_0^c)),\ y) \\ \quad (x,\ y) = g(t^a(s,\ 64) \ll o_0^c,\ t_b(s,\ 64) \ll o_0^c,\ m_0,\ 0) & \text{for } s = 0 \\ f(t^a(s,\ 64),\ t^b(s,\ 64), m_s,\ ca_s) & \text{for } 0 < s < p_B^{i^c+n} - B_c^0 \\ ((c_{B_s^c} \mathbin{\&} m(o_s^c)) \mid (x \mathbin{\&} \sim m(o_s^c)),\ y) \\ \quad (x,\ y) = g(t^a(s,\ p^b_{i^c+n}),\ t^b(s,\ p^b_{i^c+n}),\ m_s,\ ca_s) & \text{for } s = p^B_{i^c+n} - B_0^c \end{cases} \tag{5.25}$$

When $B_0^c = p^B_{i^c+n}$:

$$c_{B_0^c} := (c_{B_0^c} \mathbin{\&} \sim (m(o_0^c, o_n^c))) \mid (x \mathbin{\&} m(o_0^c, o_n^c)) \quad (x,\ y) = f(t^a(s,\ 64) \ll o_0^c,\ t^b(s,\ 64) \ll o_0^c,\ m_0,\ 0) \tag{5.26}$$

Where:

$$\begin{aligned} m(x) &= 1_2^{64} \ll x\ (= 1_2^{64-x} \parallel 0_2^x) \\ m(x,y) &= (1_2^{64} \ll x) \mathbin{\&} (1_2^{64} \gg (64-y))\ (= 0_2^{64-y} \parallel 1_2^{y-x} \parallel 0_2^x) \end{aligned}$$

and

$$t^{ar}(s,n) = \begin{cases} ar_{B_s^{ar}} \gg o_s^{ar} & \text{if } o_s^{ar} + n \leq 64 \\ (ar_{B_s^{ar}} \gg o_s^{ar}) \mid (a_{B^{ar}_{s+1}} \ll (64 - o_s^{ar})) & \text{otherwise} \end{cases}$$

which takes n bits from array ar starting in B_s^{ar} at o_s^{ar}, making sure that the first bit starts at position 0.

5.4 Scanning through arrays

This function scans through (parts of) an array. We define:

$$scan(a, i, n, g, arg) :: int? \tag{5.27}$$

Where a is the target array, i is the starting index, n is the number of elements to scan after i, g is a callback function which processes the elements, and arg is a generic argument passed to g that can be used to track task-related information. g is defined as follows:

$$g(i, v, arg) :: (arg', bool) \tag{5.28}$$

Where i is the index of the current element, v is its value, and arg is the value passed to the *scan* function. If g returns *true* at any point, the *scan* function stops and returns the current element's index. If g never returns *true* for any of the processed elements, the function returns *null* (meaning nothing), which corresponds to the question mark ? in the return type, meaning that it's optionally defined.

This function can be used for a variety of tasks, such as:

- Finding a specific value (where arg could be used to store the value to be found, and the g function returns *true* when the value has been found)

- Calculating a sum (where arg can hold the sum, and the g function always returns $false$)
- Finding the number of elements with a certain property, such as odd numbers (where arg could hold the number of elements)

We define the function:

$$scan(a, i, n, g, arg) = \begin{cases} null & \text{if } n = 0 \\ i & \text{if } n > 0 \text{ and } res = true \\ scan(a, i+1, n-1, g, arg') & \text{otherwise} \\ & \text{where } (arg',\ res) = g(i, t(a, i, bpn_a), arg) \end{cases} \tag{5.29}$$

Where:

$$t(ar, i, bits) = \begin{cases} (ar_{p_i^B} \gg p_i^b)\ \&\ m^{ar} & \text{if } p_i^b + bits < 64 \\ (ar_{p_i^B} \gg p_i^b)\ |\ (ar_{p_i^B+1} \ll (64 - p_i^b))\ \&\ m^{ar} & \text{otherwise} \end{cases} \tag{5.30}$$

In the code, the values of ar are cached, so that they don't have to repeatedly be retrieved.

Chapter 6

C Implementation

Several functions are used to allocate, edit, access, and free arrays, which correspond to the definitions of previous sections.

6.1 Allocation

The function

```
ArrayDesc *alloc_desc(uint8_t num_bits, size_t dim, ...)
```

is used to allocate an array descriptor, which stores information about an array. The data itself can be allocated using this descriptor.

- `num_bits` is the number of bits per value
- `dim` is the dimensionality of the array (= the number of dimensions).
- After this, there are `dim` `size_t` arguments, specifying the size of each dimension.

Suppose you want a 3 by 4 matrix, with 5 bits per value. Its descriptor would be allocated using `alloc_desc(5, 2, 3, 4)`.

In section 5.3 it was discussed that a redundant bit is needed to support addition in batches. This redundancy can be added with the following function:

```
ArrayDesc *add_redundancy(ArrayDesc *desc, uint8_t redundant_bits)
```

which adds a given number of redundant bits to an array descriptor, and returns that same descriptor, so that it can be coupled as such:

```
ArrayDesc *desc = add_redundancy(alloc_desc(2, 1, 3), 1);
```

Which allocates an array of two bits per element, size 3, with 1 redundant bit per element.

The functions

```
void *malloc_array(ArrayDesc *desc)
void *calloc_array(ArrayDesc *desc)
```

are used to allocate a data array corresponding to a given descriptor, using C's standard `malloc` and `calloc` functions respectively. Its type `void *`, which is a `void` pointer, is a pointer that can directly be treated as any other pointer type, allowing the functions below to remain generic instead of having to write versions for each integer data type.

The function

```
void free_desc(ArrayDesc *desc)
```

is used to free an array descriptor. The standard `free(o)` function provided in C should not be used, since the descriptor points to data that also needs to be freed.

6.2 Access

In the family of functions

```
uint8_t array_get8(const ArrayDesc *desc, const uint8_t *data, size_t index)
uint16_t array_get16(const ArrayDesc *desc, const uint16_t *data, size_t index)
uint32_t array_get32(const ArrayDesc *desc, const uint32_t *data, size_t index)
uint64_t array_get64(const ArrayDesc *desc, const uint64_t *data, size_t index)
```

each returns a single element of the array, however all return different integer types, to fit a user's needs. Ideally, the smallest integer type that can hold an element should be used, so three bits per element should use `array_get8`, 13 should use `array_get16` and so forth. It is also possible to use the larger functions for smaller datatypes, but this only uses more memory.

If the array has redundant bits that combined with the element bits is larger than a type, but without redundancy the elements still fit in the data type (for instance eight bits per element with one redundant bit), the smaller data type can still be used (in this case `array_get8`), since the value fits in this type.

`index` is a single number. If the array has a higher dimensionality, the following function can be used to calculate the correct index value for a given position:

```
size_t get_index(const ArrayDesc *desc, ...)
```

which accepts the array descriptor, followed by a `size_t` for each dimension of the array.

Suppose you have a 2-dimensional array, and you want to read the element at position 2, 6. With a C array, you would use `array[2][6]`. Instead, you should now use `get_index(desc, 2, 6)` and pass this to an `array_getX` or `array_setX` variant.

The functions

```
uint8_t array_set8(const ArrayDesc *desc, uint8_t *data, size_t index, uint8_t val)
uint16_t array_set16(const ArrayDesc *desc, uint16_t *data, size_t index, uint16_t val)
uint32_t array_set32(const ArrayDesc *desc, uint32_t *data, size_t index, uint32_t val)
uint64_t array_set64(const ArrayDesc *desc, uint64_t *data, size_t index, uint64_t val)
```

are all used to set a single element in the array to a given value.

6.3 Bulk operations

The following functions each have several versions, for various levels of optimization.

A big difference with the `get_X` and `set_X` functions from before, is that indices and counts specified here are multi-dimensional arrays (`const size_t *`) rather than single `size_t`s. This is due to the nature in which the operations work, and provides the best performance.

6.3.1 Filling with one value

The function

```
void batch_fill(const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count,
     uint64_t val)
```

is used to fill (part of) an array with a specific value, and corresponds to section 5.1.

`offset` specifies the offset at which to start filling. If this is `NULL`, it defaults to zero in all dimensions.
`count` specifies the number of elements in each dimension to fill. If this is `NULL`, it defaults to the array's size.
`val` specifies the value to insert

`count` cannot be `NULL` if `offset` is not zero, since then out-of-bounds access occurs.

6.3.2 Filling with varying values

The function

```
void batch_set(
       const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count,
       uint64_t (*action)(const size_t *index, void *arg), void *arg
)
```

is used to fill (part of) an array with dynamically generated values, and corresponds to section 5.2.

- `action` is a function reference, which is given an `index` and the originally provided `arg`, and should return the desired value for the index.
- `arg` can be any pointer, and is passed to `action`. Since C does not support objects, this is a way of passing context to the `action` function.

6.3.3 Compute

The function

```
void batch_op(
       const ArrayDesc *dx, const void *x, const size_t *ox,
       const ArrayDesc *dy, const void *y, const size_t *oy,
       const ArrayDesc *dz, void *z, const size_t *oz,
       const size_t *count,
       uint64_t (*action)(uint64_t x, uint64_t y, uint64_t *carry, uint64_t mask, void *arg),
            void *arg
```

```
)
```

is used to compute values using two input arrays (`x` and `y`), and stores the result in an output array (`z`). This corresponds to section 5.3.

The three arrays all need to have the same number of bits (including redundancy). But their size is allowed to vary, hence why all three need to have their descriptors passed along as well.

- `action` is a function reference, which is given the `x` and `y` value, a reference to a `carry` which can be passed on to the next computation (useful for addition), a `mask` which should be used to mask the output value, and the `arg` that was passed to `batch_op`.

6.3.4 Scan

The function

```
size_t *batch_scan(
       const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count,
       bool (*action)(const size_t *index, uint64_t val, void *arg), void *arg
)
```

is used to scan through a given array, and corresponds to section 5.4. `action` is called for each element, and the function stops if `action` returns true or all elements have been scanned. Note that unlike in section 5.4, the `action` function only returns a `bool`, and the `arg` parameter is mutable, which slightly differs from the definition but allows for the same behaviour.

- `action` is a function reference, which should return `true` if the scan should stop. It's given an `index`, a `val` (which is the value at the index), and `arg` is the pointer originally provided to `bulk_scan`.

Chapter 7

Performance

In order to see the performance cost or benefit of using aforementioned functions, tests have been set up to compare each function against a regular C implementation.

7.1 Tasks

The following tasks have been set up to focus on the performance of various functions.

- `sum`: The sum is computed of an array's elements. This stresses read performance and can take advantage of the `batch_scan` function.
- `fill`: A given array is filled with a single value. This stresses write performance and can take advantage of the `batch_fill` function.
- `counter`: In a given array a, for each index i, a_i is assigned the value i (truncated to fit the number of bits). This stresses write performance and can take advantage of the `batch_set` function. This task is designed to mimic the `EvenOdd` benchmark in C. Grelck and H. Luyat[1], to see how performance compares.
- `xor`: Two input arrays a, b and an output array c of identical size are given. For each index i, $c_i := a_i$ XOR b_i, which is the bitwise XOR function, that per bit produces `1` if the two input bits differ, and `0` otherwise. This does both reading and writing, and can test the performance of the `batch_op` function.
- `add`: Similar to `xor`, except additions are performed rather than a bitwise XOR. This also allocates an extra redundant bit needed for addition, as discussed in section 5.3.
- `gauss`: This is the most complex benchmark. A window of 11 elements is moved across an input array. On each window, a simple operation is performed to produce a single output value. The result is then stored in another array. Since this is a less straight-forward task, it can make less use of the `batch` functions, and is used to see what performance can be accomplished. This task is designed to mimic the `Gauss` benchmark in C. Grelck and H. Luyat[1], as another way to compare results to their paper.

7.2 Optimization levels

Each of these have several variations, to test various levels of optimization:

- `standard`: A standard implementation, using normal C arrays and corresponding calls.
- `compact`: A compact implementation, simply replacing the standard operations with their compact counterpart function. For instance, `array[i]` becomes `array_get8(descriptor, array, i)`. This allows for a drop-in replacement for existing code, but has little room for optimizations.
- `dynamic`: An implementation using compact array storage, with functions that can perform tasks in batches over larger chunks of data, which have been described in chapter 5. Being the most flexible, the functions used in these tests:
  1. support multidimensional arrays (any number specified by the user).
  2. are generic, in that they have a parameter that is a function that calculates resulting values (see section 6.3), which prevents having to make a unique version of the batch function for various use-cases, such as adding. The downside is that there is less room for the compiler to optimize, and on top of that, more function calls typically means worse performance.
  3. allow the number of bits per element in an array to be specified at run-time.
- `generic`: An implementation using compact array storage, the same as `dynamic`.

  As opposed to point 3 however, the number of bits per element in an array is this time fixed at compile-time, which reduces flexibility but allows for more compiler optimizations.

  Note that the `alloc_desc` function described in section 6.1 still takes the `num_bits` argument, however this value is simply ignored this time, and completely untouched during compile-time. The reason it's still included is to make it easier to compile different versions of the code, and since internally no code exists relating to the `num_bits` value, there is no performance impact from keeping the argument.
- `specialized`: An implementation similar to `generic`, but with specialized functions, so that there is a unique function for the to-be-run tasks (such as `gauss`, `add`). This reduces the number of function calls and keeps more code in one place, which in turn allows for more optimizations to be done by the compiler.

  The downside is that new versions need to compiled of the required function for various use-cases, but this can be alleviated by turning the function into a #define rather than a function directly, which means that the function can be generated dynamically, but still in the end be a single function after compiling.
- `optimized`: An implementation akin to `specialized`, that is further optimized so that only 1-dimensional arrays are supported, replacing offsets' and counts' use of `const size_t *` with just `size_t`. This is the version that has the minimum amount of code, and so should be close to the theoretical optimization limit that can be achieved with the implementation I made.

7.3 Sizes

These tests have been performed for:

- arrays of 100 and 100,000 elements, to measure how performance scales with array size.
- arrays with 1, 2, 5, 10 and 11 bits per element, to measure what performance various bits per element can accomplish.

7.4 Specialized functions

The following functions have been added for the `specialized` and `optimized` instances:

Since `batch_fill` is already specialized, only an optimized version exists:

```
void batch_fill_opt(const ArrayDesc *desc, void *data, size_t offset, size_t count, uint64_t
    val)
```

There is a specialized along with an optimized version of `batch_set` for the `counter` task:

```
void batch_set_index(
       const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count
)
void batch_set_index_opt(
       const ArrayDesc *desc, void *data, size_t offset, size_t count
)
```

There are specialized and optimized versions of `batch_op` for the `xor` and `add` tasks:

```
void batch_op_xor(
       const ArrayDesc *dx, const void *x, const size_t *ox,
       const ArrayDesc *dy, const void *y, const size_t *oy,
       const ArrayDesc *dz, void *z, const size_t *oz,
       const size_t *count
)
void batch_op_add(
       const ArrayDesc *dx, const void *x, const size_t *ox,
       const ArrayDesc *dy, const void *y, const size_t *oy,
       const ArrayDesc *dz, void *z, const size_t *oz,
       const size_t *count
)
void batch_op_xor_opt(
       const ArrayDesc *dx, const void *x, size_t ox,
       const ArrayDesc *dy, const void *y, size_t oy,
       const ArrayDesc *dz, void *z, size_t oz,
       size_t count
)
void batch_op_add_opt(
       const ArrayDesc *dx, const void *x, size_t ox,
       const ArrayDesc *dy, const void *y, size_t oy,
       const ArrayDesc *dz, void *z, size_t oz,
       size_t count
)
```

Lastly, for the `sum` and `gauss` operations, dedicated functions have been added. The `batch_scan_gauss` function doesn't perform the whole task, but instead is responsible for the operation performed on a window, and the result is then written to an output array:

```
uint64_t batch_scan_gauss(
       const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count
)
uint64_t batch_scan_sum(
       const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count
)
uint64_t batch_scan_gauss_opt(
       const ArrayDesc *desc, void *data, size_t offset, size_t count
)
uint64_t batch_scan_sum_opt(
       const ArrayDesc *desc, void *data, size_t offset, size_t count
)
```

7.5 Environment

All tests have been performed on the following hardware:

- Dell XPS 15 9510 (0A61)
- Intel Core i7-11800H (8 cores, 16 threads) @ 2.30GHz
- 2x8GiB of SODIMM DDR4 Synchronous @ 3200MHz (0.3 ns latency)

Running the following system:

- Fedora Linux 35 64-bit (Workstation Edition)
- Linux kernel 5.19.12-100.fc35.x86_64
- KDE Plasma version 5.25.4 (KDE Frameworks version 5.96.0)
- Graphics Platform X11

In order to obtain accurate timings, these benchmarks use CPU time rather than real time. Real time can be influenced by other processes running on the computer, which makes it difficult to obtain accurate results not affected by these. CPU time is the time spent by the program on the processor, which is a much more precise measure of code performance.

Laptops often use frequency scaling to save power, which makes CPU cores perform slower at times. In order to obtain consistent benchmark results, the command `cpupower frequency-set -g performance` has been used to disable this.

The C compiler that was used, `gcc`, has various compilation optimization levels in order to try to make a program perform quicker, achieved by removing and simplifying instructions, while ensuring that the program still behaves the same. These levels go from O0 to O3, where O0 means no optimization, and O3 means the most optimizations applied. In the real world, O3 is always used, which is why the tests have been run with O3.

7.6 Results

In order to obtain an accurate average performance measurement, each task is first run 1000 times as a warm-up, and after that 10,000 executions are done, and the average run-time of these are taken as the benchmark. Measures have been taken so that on each execution slightly different data is read and written, so that the program cannot do any unwanted optimization regarding data re-use.

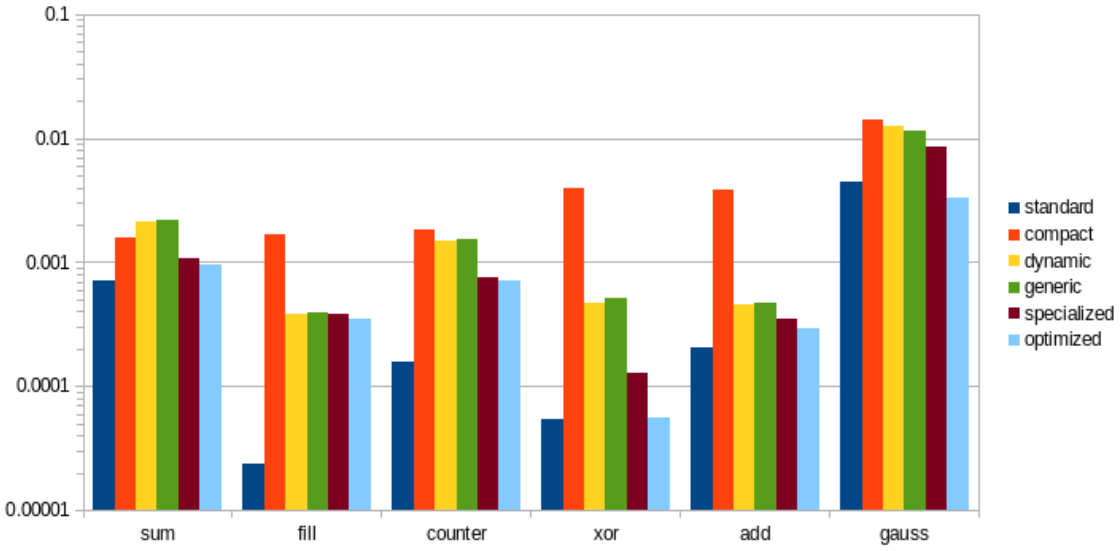


Figure 7.1: Runtime in seconds (logarithmic), 1-bit elements, size 100

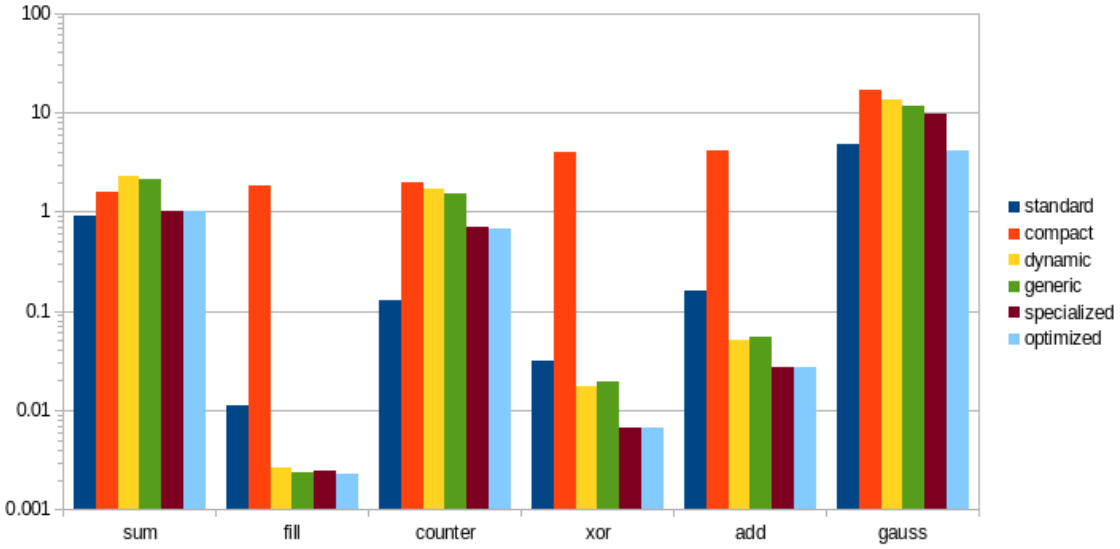


Figure 7.2: Runtime in seconds (logarithmic), 1-bit elements, size 100,000

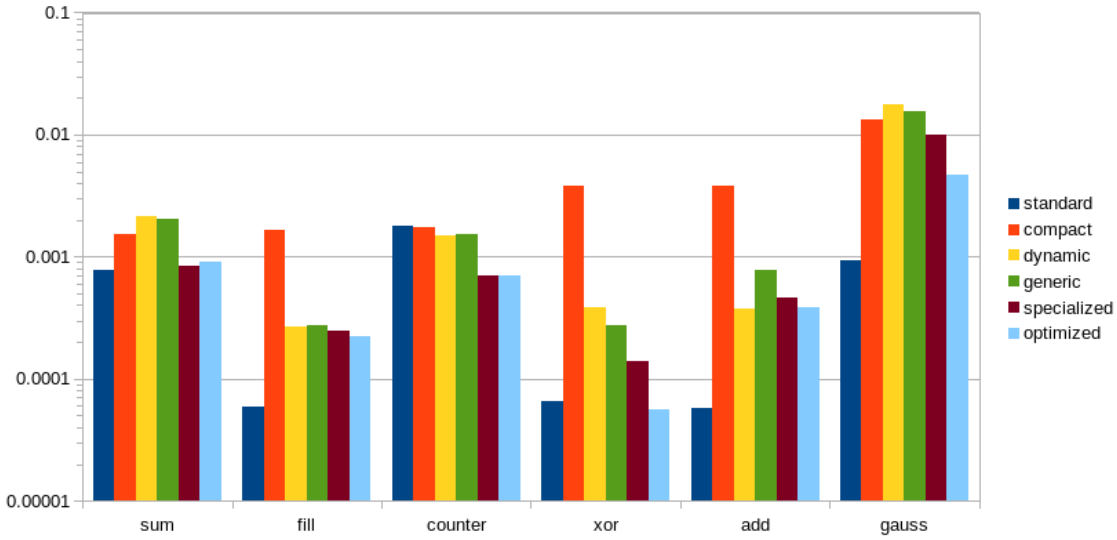


Figure 7.3: Runtime in seconds (logarithmic), 2-bit elements, size 100

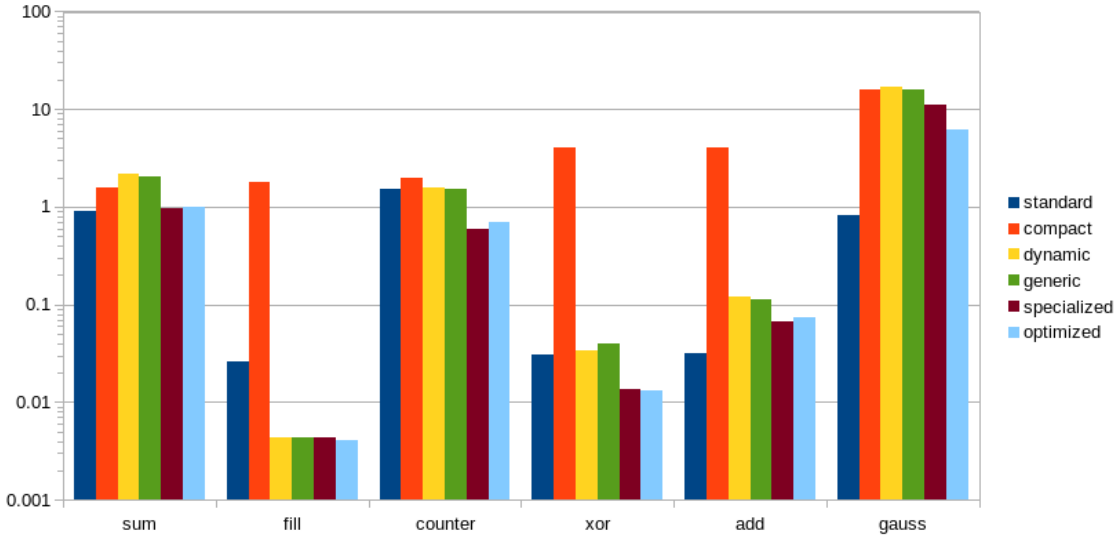


Figure 7.4: Runtime in seconds (logarithmic), 2-bit elements, size 100,000

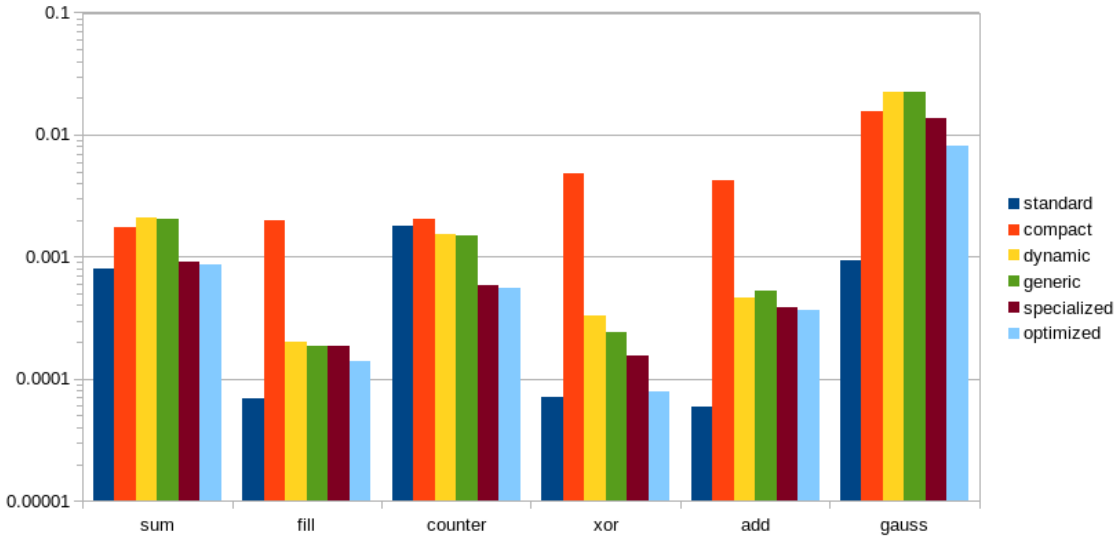


Figure 7.5: Runtime in seconds (logarithmic), 5-bit elements, size 100

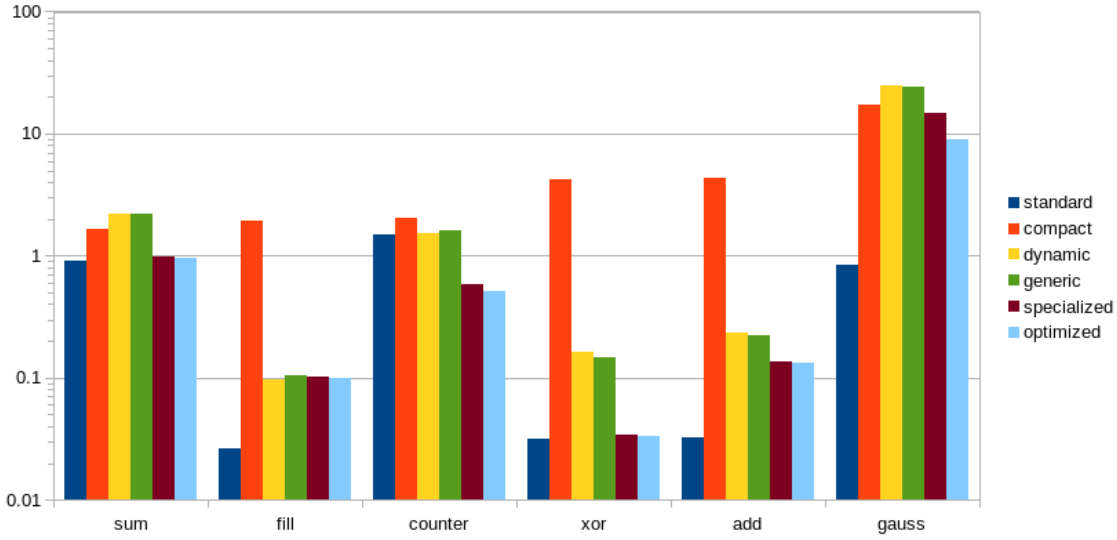


Figure 7.6: Runtime in seconds (logarithmic), 5-bit elements, size 100,000

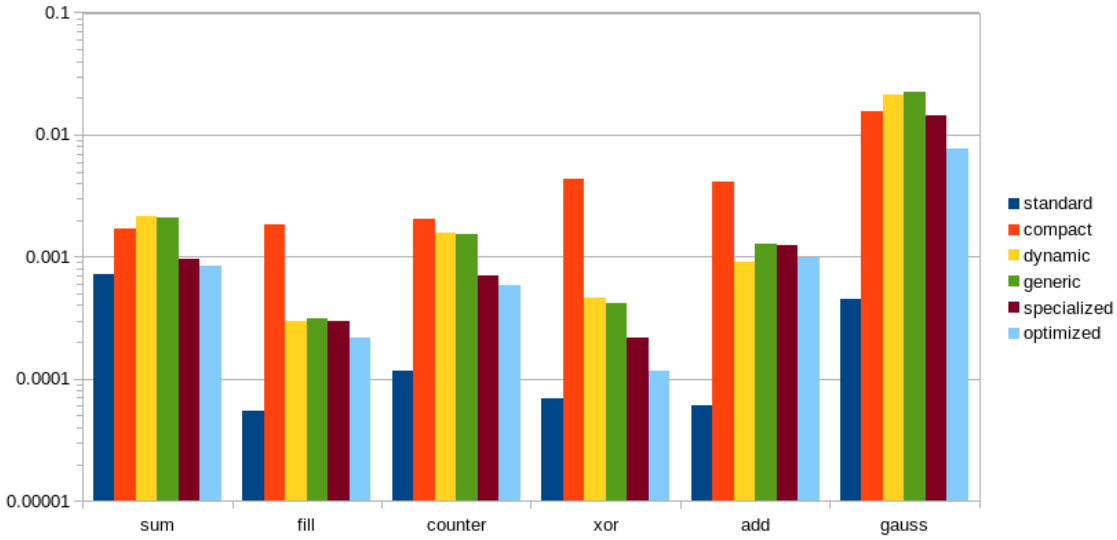


Figure 7.7: Runtime in seconds (logarithmic), 10-bit elements, size 100

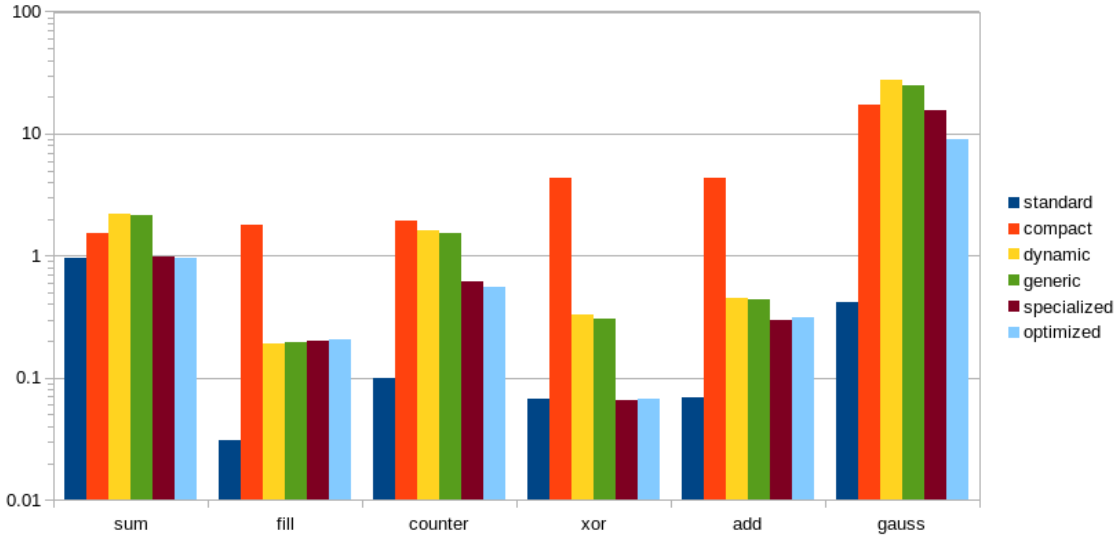


Figure 7.8: Runtime in seconds (logarithmic), 10-bit elements, size 100,000

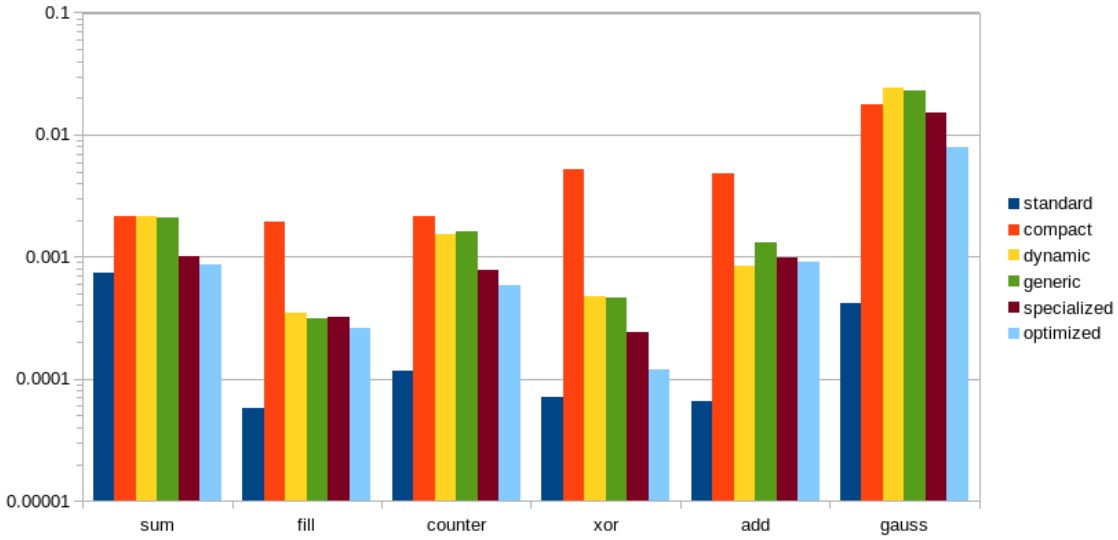


Figure 7.9: Runtime in seconds (logarithmic), 11-bit elements, size 100

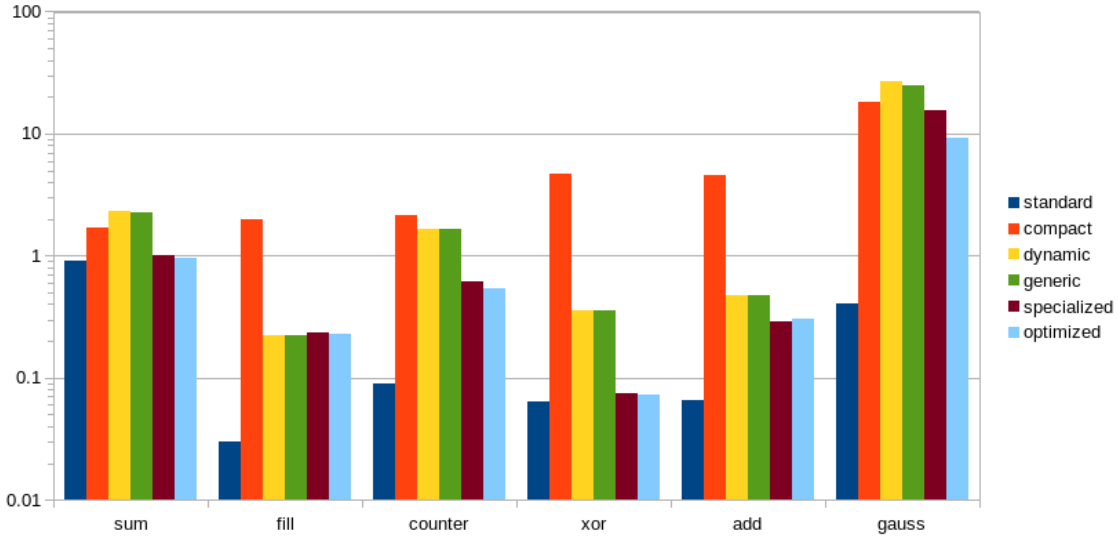


Figure 7.10: Runtime in seconds (logarithmic), 11-bit elements, size 100,000

7.6.1 Analysis

Since performance can vary quite a bit, the graphs are shown on a logarithmic scale, so that each major line differs by a factor of 10. There are ticks on the side of the graphs which show smaller steps as well.

In some cases, the standard implementation seemed to perform faster in some benchmarks in higher bit counts, which of course should not be happening. I assume there is some optimization going on that should not be happening, probably relating to the C compiler doing repeated tasks faster, but I could not find the cause of it nor prevent it. Therefore, those results should be taken with a grain of salt, and what should be considered more important is how the non-standard implementations compare to the same benchmark in lower bit counts. However, just because some extra optimization is going on, the results are still a very good indicator of what to expect.

We start by looking at the results for 1-bit arrays:

- Overall, the `compact` implementation seems slowest, which makes sense considering it's not very optimized.
- The `dynamic` and `generic` implementations perform equivalently to `compact` in the `sum`, `counter` and `gauss` benchmarks. In the rest of the benchmarks, these two implementations do reach some speed-up, particularly in the `fill` task. Granted, with smaller arrays these functions are not as fast as a standard implementation, so the performance improvement only occurs at larger arrays.
- The performance difference between `dynamic` and `generic` is negligible, so there seems to be no benefit to defining arrays' number of bits per element at compile-time rather than run-time. This is good news, because it means that no separate versions of the program need to be compiled to handle arrays with a different number of bits per element.
- Moving to the `specialized` implementation, its performance seems much more promising overall. In the `fill` and `gauss` tasks its performance doesn't differ much from `dynamic`, which is because the `fill` function is already specialized, and `gauss` cannot be done with a single function call; related batch functions need to be called for each window of 11 elements. But the `counter` and `xor` tasks are noticeably faster compared to the `generic` implementation independent of array size, and `sum` if quite a bit faster in larger arrays.
- The `optimized` function has very similar performance to `specialized` in the a large array, but with a smaller array it tends to be a bit faster. This is probably since the `optimized` functions don't need to do as much initialization and dimension tracking as the other functions need to do. So when the array size is small, the extra time needed for initialization is more noticeable. Only `gauss` seems to perform slightly better optimized. This is likely due to having to call one of the batch functions for each window, and with less initialization in every call, overall there can be quite some time saved.

With 2-bit arrays, the `fill` function is still noticeably faster, `gauss` is a bit slower, and the rest are on par with a standard implementation. So although it's slightly worse, it's still reasonable to use this compact version without losing much time.

Moving up to 5-bit arrays, where only 3 bits are saved opposed to an entire byte, performance is less favourable. `sum`, `counter` and `xor` can still keep up with the standard, but the other functions are now slower. The reason for this shift is most likely due to the fact that the least common multiplier of 5 and 8 is 40, which means that a lot of values are spread across bytes and more data shifting needs to be done when reading and writing.

In 10-bit arrays, `xor` can keep up with the standard, but the other functions are now noticeably slower. It seems at higher bit counts, not much time can be saved.

At 11 bits, where again there is not a small common multiplier, performance is also less ideal. Although interestingly, it's still very similar to 10-bit arrays. It appears that at this point the data shifting does not make a big difference. It's likely that memory throughput is the issue in this scenario.

7.6.2 Discussion

What can be deducted from these results, is that specialized functions are needed to achieve performance that is similar to or better than a standard implementation. Using a generic function often can quite drastically worsen performance, so using those is not recommended unless performance is not an issue.

In the low bit counts, where the difference in element size between the standard and custom implementation is largest, the performance is most favourable. With increased bit counts the time penalty is larger, and there is no real benefit to use such a library if performance matters. And it is probably smart to stick to numbers that divide byte sizes, such as 1, 2 and possibly 4.

Comparing to the results of C. Grelck and H. Luyat[1], the outcomes are very similar with matching tasks. Their Gauss test, which stresses read performance, sees identical runtime between compact and standard implementations. In my `gauss` version, which is designed to mimic theirs, the `specialized` code can reproduce those results. Looking at the outcomes for my `sum` function, which also stresses reads, performance of the compact representation again appears to be very close to the standard in the `specialized` instance.

Looking at write tasks, my `counter` function is most similar to their `EvenOdd`. Both stressing write performance, at a single core, run-time is always slightly slower in a dense representation. So in this scenario, results seem on par with that of their paper. But when using the specialized faster functions `fill`, `xor` and `add`, the dense representation actually saw a noticeable time-save. This shows that by combining standards reads and writes with such batch operations, it is possible to save a noticeable amount of time performing integer-related tasks.

Chapter 8

Related Work

8.1 Sub-Byte Storage

C. Grelck and H. Luyat[1] have compared performance between traditional 8- and 32-bit representations for booleans in arrays, and storing booleans on a per-bit basis. Using SaC (Single Assignment C), they found that on a single core, performance becomes nearly identical between implementations in read scenarios, and towards 32 cores, the dense implementation approaches around a 2-fold performance improvement. In write scenarios, performance of dense arrays is very similar to their sparse counterparts in higher core counts, but single core performance is 5 to 6 times worse.

The great performance, as mentioned in the paper, is due to caching of data read from memory, which outweighs the performance loss from having to shift data around for reading and writing. This shows that there is potential in using dense arrays when used properly, but there still remains research to be done on more specialized functions, and also on other integer sizes than one bit.

Klages et al.[2] researched using 4-bit integers in computations, and devised a way to perform multiplication and addition in a single fused instruction, for multiple values at once using 32-bit registers. With this, they could achieve an effective 5.6 TOPS (Tera Operations Per Second) on a 4.3 TOPS GPU.

M. Hayes[3] created a way to use conventional paradigms in Fortran-77 to manipulate integers smaller than a byte without having to worry about the underlying architecture or implementation. This is a paper from 1986, which shows that interest in compact integer storage has been around for a long time.

Duncan et al.[4] tackled the problem of storing smaller-than-byte integers in network packets and communicating them between systems that may differ in architecture. Specifically, their endianness may differ, which has implications for the internal structure.

These two papers have created ways to make it easier to incorporate smaller-than-byte integers into everyday code, which shows that interest in the subject has been around for a long time, and that it's possible to make an interface in a way that is programmer-friendly.

8.2 Integers in Neural Networks

Shuang et al.[5] researched using low-bitwidth integers in the training of neural networks. Traditionally this is done using floating point numbers, and prior research had already been done into using lower precision numbers, however this paper sought to switch to integers entirely, which results in comparable accuracy while providing higher energy efficiency. Neural networks like these could greatly benefit from using a compact integer storage method, since networks can often contain thousands of nodes, and if data could be saved for each of these, memory usage could be noticeably reduced. Combine that with specialized functions for important operations, and similar performance could be achieved with a reduced memory footprint.

Das et al.[6] also did research into integer training of neural networks, particularly in larger networks, and saw identical or improved accuracy compared to the state-of-the-art while providing a 1.8X improvement in end-to-end training throughput.

Maolin et al.[7] did more research on integer storage in neural networks and achieves negligible accuracy degradation on the MNIST, CIFAR10, and ImageNet datasets compared to using floating points.

The above three papers show that integers can be used effectively for neural network training, and show a potential use-case for the results found in this paper. More work needs to be done surrounding making functions specifically designed for neural-network training, but there is promise for memory savings without much performance degradation, with there even being ways to save time.

Trusov et al.[8] made an efficient implementation for quantized 4-bit matrix multiplication. Quantized integers have been shrunk down compared to normal floating point integers, so that they consume significantly less memory, and in this case, only 4 bits. It "shows 2.9 times speedup compared to standard floating-point multiplication and is 1.5 times faster than 8-bit quantized one".

T. Dettmers and L. Zettlemoyer[9] researched using 3 to 8-bit integer sizes in LLMS, measuring their zero-shot performance. The conclusion is that 4 bits is almost universally optimal for total model bits and zero-shot accuracy, which shows that this is the most likely bit size to use in these networks.

Chapter 9

Conclusion

Using fewer bits per element in arrays is an interesting topic that can be a very effective way to save data. Making sure it performs well becomes more difficult as operations increase in complexity, but it is clear that it's possible to achieve performance on par with standard implementations.

What's more, with the option of special batch operations, such as the `fill` and `batch_op` functions, it's even possible to save time, even on smaller arrays. So it's clear that this area could offer a way to save memory as well as time when working with integer arrays.

Although at larger bit counts it's not as easy to achieve similar performance, at smaller values the performance benefit is very noticeable, so focussing on that in future work is recommended.

9.1 Future work

Currently, the generic functions need actions passed along to execute the task at hand. A mid-ground solution for reducing duplicate code while still having specialized functions, is to change these from pre-defined functions, to #defines. For instance, the `batch_set` function can be changed to:

```
#define define_batch_set_function(function_name, action) void function_name( \
        const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count \
) { \
// ...
```

and further down, uses of the `action` function can be replaced with this. This makes a unique function for each operation, but no intermediate function calls are done, so this can achieve performance akin to `specialized`. For instance:

```
define_batch_set_function(batch_set_index, index)
// Would produce:
void batch_set_index(
        const ArrayDesc *desc, void *data, const size_t *offset, const size_t *count
) {
// ...
```

What would be interesting to investigate, is a function that could perform the entire `gauss` task in one function. It is possible to create an aggregation function, where windows of input values can be transformed into single output values, that could bundle writes to the output array together. Such a function could cover the gauss problem, but also a larger collection of problems, such as local sums and additions. I imagine it with the following signature:

```
void batch_aggregate(
       const ArrayDesc *dx, const void *x, const size_t *ox, const size_t *sx, const size_t *
           wsx,
       const ArrayDesc *dy, void *y, const size_t *oy, const size_t *sy,
       const size_t *window_size, const size_t *count,
       uint64_t (*init)(void *arg),
       uint64_t (*action)(uint64_t x, uint64_t y, uint64_t value, void *arg),
       void *arg
);
```

Where `dx`, `x` and `ox` have the same role as in other functions, `sx` is the step size: the distance between each window's starting position, and `wsx` the distance between each element within a window. `window_size` specifies the number of steps within a window, `init` is used to obtain the starting value for the window, and `action` is the update step.

I imagine as a `dynamic` or `generic` version it might not be able to provide great performance, but if it was specialized, it might be able to perform even faster than the current non-standard `gauss` implementations.

Bibliography

[1] C. Grelck and H. Luyat, "Towards truly boolean arrays in data-parallel array processing," *Advances in Parallel Computing*, vol. 25, pp. 82–91, 2014. [Online]. Available: https://doi.org/10.3233/978-1-61499-381-0-82.

[2] P. Klages, K. Bandura, N. Denman, A. Recnik, J. Sievers, and K. Vanderlinde, "Gpu kernels for high-speed 4-bit astrophysical data processing," 2015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7245729.

[3] M. Hayes, "Accessing bit fields in fortran-77," 1986. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/6617.6620.

[4] R. Duncan, P. Jungck, and D. Mulcahy, "Portable bit fields in packetc," 2011. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4302-4159-1_34.

[5] W. Shuang, L. Guoqi, C. Feng, and S. Luping, "Training and inference with integers in deep neural networks," 2018. [Online]. Available: https://arxiv.org/abs/1802.04680.

[6] D. Das, N. Mellempudi, D. Mudigere, *et al.*, "Mixed precision training of convolutional neural networks using integer operations," 2018. [Online]. Available: https://arxiv.org/abs/1802.00930.

[7] M. Wang, S. Rasoulinezhad, P. H. Leong, and H. K. So, "Niti: Training integer neural networks using integer-only arithmetic," 2020. [Online]. Available: https://arxiv.org/abs/2009.13108.

[8] A. Trusov, E. Limonova, D. Slugin, D. Nikolaev, and V. V. Arlazarov, "Fast implementation of 4-bit convolutional neural networks for mobile devices," 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9412841.

[9] T. Dettmers and L. Zettlemoyer, "The case for 4-bit precision: K-bit inference scaling laws," 2022. [Online]. Available: https://arxiv.org/abs/2212.09720.