Reducing the step count of 256-bit operations in Cairo

Yevgeny Zaytman

Follow

Nethermind.eth

--

Listen

Share

by Yevgeny Zaytman and Albert Garreta

Special thanks to

[Ignacio Manzur

](https://www.linkedin.com/in/ignacio-manzur-0220121b7)and

[NonCents

](https://twitter.com/0xNonCents)for their help with Cairo.

Cairo's basic data type, the felt

, has capacity for storing (non-negative) integers of up to 251 bits. However, many applications require managing 256-bit integers. As a result, Cairo's standard library includes a sub-library that handles such functionality.

In this post, we present several improvements we have made in the implementations of these functions. We have generally reduced the number of steps and range check calls by about 25–45% in most functions. This improvement is greater in some cases (like in the function that computes square roots).

Context

Before we begin, let us review some Solidity 101.

Solidity's default data type, Uint256

, represents integers between 0 and 2^{256}–1, i.e., integers that can be written with at most 256 bits. Being Solidity's default, this data type is ingrained in the whole EVM-based blockchain space. A prominent example (among many) is the ERC20 token interface, which works almost exclusively with Uint256

types.

On the other hand, Cairo has one and only one data type, called felt

(an abbreviation for "field element"). This data type is used to represent integers between 0 and p

–

1, where p =

$2^{251} + 17 \cdot 2^{192} + 1$ is a prime of 252 bits

. Hence, a felt

can store all positive integers of up to 251

bits

and only a "bunch" of 252-bit integers.

All Cairo developers are fully aware of this discrepancy between Solidity's and Cairo's data type, and, indeed, the "251 vs. 256" difference has been the source of several memes within the community. On a more serious note, this discrepancy can be the root of some headaches, since many StarkNet projects must work with a 256-bit data type for L1←→L2 compatibility.

To allow working with 256-bit integers, Cairo's standard library includes adedicated submodule, in which a 256-bit integer n

is represented as two 128-bit integers (sometimes referred to as chunks

or limbs

), n_0 and n_1 (called n.low

and n.high

in the library), so that $n = n_0 + 2^{128} n_1$. In other words, n_0 is the positive integer formed by the 128 least significant bits of n

, and n_1 is the positive integer formed by the rest of the bits. Having 128 bits, each one of these chunks is small enough to be stored in a single felt. The two resulting felts are stored together within a 2-felt struct called Uint256

.

So far, so good, but much work still needs to be done. Indeed, suppose you have stored two 256-bit integers n, m

in this fashion. How can we now compute their product $n \cdot m$

in Cairo?

In principle, $n \cdot m$

may have up to $2 \times 256 = 512$ bits. Hence one generally needs two Uint256

instances to store $n \cdot m$

(or, equivalently, four 128-bit chunks). The standard method for doing this is similar to what we did before: split the 512-bit integer $n \cdot m$

into two 256-bit chunks C_1, C_2, so that $n \cdot m = C_1 + 2^{256} C_2$. Both C_1, C_2 will be stored as Uint256

instances.

The main question here is: how do we efficiently obtain C_1 and C_2, given that we only know n_0, n

$_1$, m

$_0$, m

$_1$? There is not a clear definitive answer to this, and it is here where the bulk of our work resides.

So far, we have discussed only the multiplication of 256-bit integers, but similar issues arise with all other operations that involve 256-bit integers: addition, subtraction, squaring, taking a square root, etc.

Contributions

We next outline our contributions. We discuss exclusively 256-bit (non-negative) integers, but most improvements can be adapted to integers of other sizes (384, 512, etc.)

Multiplication

We have implemented a new method for multiplying two 256-bit integers. Compared to the corresponding function from Cairo's standard library, our function is approximately 31% cheaper in Cairo steps and 36% cheaper in "range check builtin" calls:

Link to code

Division

The multiplication improvement percolates into other functions such as division with remainder. In Cairo, a simple method to obtain the quotient and the remainder of dividing an integer n

by another integer m

is to:

1. Compute the quotient q

and the remainder r

in a hint.

1. Verify in Cairo (outside the hint), that n = m

·q

- r

and 0 ≤ r < m

.

Implementing this strategy using our new multiplication function (with some extra minor modifications) results in the following efficiency improvements for the division function in Cairo's standard library.

Link to code

Squaring

Since many number theoretic algorithms make extensive use of squaring (i.e., computing n

² for n

a 256-bit integer), it is worthwhile to create a function to handle this particular case. The standard library performs squaring by calling the multiplication function with input n, n

. We have improved upon that by building a dedicated squaring function, which, together with our new multiplication function, results in efficiency improvements of about 50%.

Link to code

Taking the square root

We also improved the efficiency of taking a square root by 1) using a 128-bit squaring function and 2) removing one of the two felt multiplications the current function uses.

Link to code

Subtraction

The subtraction function was also improved. We leveraged Cairo's hints to save us one addition operation.

Link to code

Uint256_expand

Finally, we created a new class called Uint256_expand

, an expanded version of Uint256

that stores the two 128 chunks of a 256-bit non-negative integer n

, together with extra data that needs to be computed when multiplying n

by another integer.

This class (which is more expensive to instantiate and consumes more memory than Uint256

) is to be used for integers n

that repeatedly appear in our operations. An example could be the scenario where we are working modulo an integer n

. In this case, we will need to divide by n

many times, and storing n

as a Uint256_expand

will save us some duplicate work.

Link to code

Relevance for cryptography implementations

Another application worth mentioning is that our techniques can be extended to integers of arbitrary size. This is of interest primarily for cryptography applications, where one often works with huge numbers, and efficiency is paramount. The BLS signature implemented over the BLS12–381 elliptic curve is a relevant example in the blockchain space. Such a scheme is native to Ethereum, and it operates with integers of 381 bits.

Stay tuned

Besides 256-bit operations, we have been working on other basic mathematical functionality such as prime field arithmetic (for large primes) and elliptic curve cryptography. Stay tuned for further updates about this work!