Max, min, and sort functions using Programmable Bootstrapping in Concrete FHE

[17 minute read]

TL;DR

In this article, we show how to compute the maximum and minimum of an encrypted array of data without decrypting it, using Programmable Bootstrapping in Concrete FHE. We also demonstrate how to implement a sorting algorithm using these operations, which can work on encrypted data. Finally, we comment on the potential for optical acceleration, showing that Gaussian technology can accelerate the main bottleneck by more than 100×.

Introduction

Fully Homomorphic Encryption (FHE) allows for arbitrary calculations to be performed on encrypted data, thus removing the need to decrypt it for analysis. In principle this gives the last remaining piece to data security: protection against forms of encryption, data only is encrypted when at rest (e.g. on a hard drive) or in transit (such as passing through the internet). FHE is the only generalizable technique that allows data to be used while encrypted.

However, FHE currently suffers from two major issues that hinder its commercial viability. The first one is speed: calculations on encrypted data can be thousands to millions of times slower than what can be achieved in plaintext data. The second is the domain knowledge required to implement and scale FHE software efficiently.

Fortunately, rapid progress has been made in the years following Craig Gentry's original theory, where the first FHE scheme was proposed. The most recent schemes (TFHE, Zama's Concrete library, and the CLEAR library written by the Google FHE team, Google's FHE transpiler is set to stream regular C++ code into TFHE-code via an XLS intermediate representation generated by Google, making it straightforward to convert existing code into FHE-compatible binaries, and Google's HEaaS offering) offer superior knowledge of the field.

Zama's Concrete library is also built on TFHE and uses programmable bootstrapping to perform any univariate operation in a single bootstrap. This significantly reduces the number of operations which needs to be performed during tasks such as encrypted deep learning inference. It also provides high level functions for homomorphic operations, these include such things as the addition of two encrypted integers, multiplication of encrypted integers, and the application of a univariate function to an encrypted integer. Using these operations but also private-public-secret keys and default parameters, allows for non-cryptographic developers to start working quickly and safely with FHE. (See the documentation for more details.)

Concrete has already been used to demonstrate several applications with both industrial and academic interest, over the course of testing our optical processing hardware design, we've performed many of our own demonstrations in tasks such as secret-key neural network inference, Convolutional Neural Networks (CNN), and encrypted string search.

In this tutorial, we'll add to these demonstrations by demonstrating how to use Concrete to implement functions which compute the maximum and minimum of an encrypted array of data, and how to use these functions to implement a sorting algorithm on encrypted data. These operations are useful for a number of reasons, for example, they are a component in certain pooling layers in Deep Learning, they are an example of the force at the end of the course, and they are used in many of our demonstrations of our optical processing hardware.

Although we are developing optical computing chips that can accelerate the main components of FHE calculations, these solutions are still early, so we are somewhat out of reflex to better understand the components of FHE and how they accelerate. We aim to develop a solution that not only works for FHE, but works as well as possible.

Beyond this, we also are trying to take advantage of working with the cryptosystem to ensure that FHE continues to develop into a tool which is as user-friendly as possible. This is especially important given that FHE touches upon deeper aspects of the application space in a way that most other tools do not, for example, the choice of cryptographic parameters can affect the accuracy of the output of a machine learning. It is essential for the success of FHE that developers, data scientists and other users have the resources to help them build this ecosystem safely and with FHE. (See the documentation for more details.)

Concrete has already been used to demonstrate several applications with both industrial and academic interest, over the course of testing our optical processing hardware design, we've performed many of our own demonstrations in tasks such as secret-key neural network inference, Convolutional Neural Networks (CNN), and encrypted string search.

In this tutorial, we'll add to these demonstrations by demonstrating how to use Concrete to implement functions which compute the maximum and minimum of an encrypted array of data, and how to use these functions to implement a sorting algorithm on encrypted data. These operations are useful for a number of reasons, for example, they are a component in certain pooling layers in Deep Learning, they are an example of the force at the end of the course, and they are used in many of our demonstrations of our optical processing hardware.

Although we are developing optical computing chips that can accelerate the main components of FHE calculations, these solutions are still early, so we are somewhat out of reflex to better understand the components of FHE and how they accelerate. We aim to develop a solution that not only works for FHE, but works as well as possible.

Beyond this, we also are trying to take advantage of working with the cryptosystem to ensure that FHE continues to develop into a tool which is as user-friendly as possible. This is especially important given that FHE touches upon deeper aspects of the application space in a way that most other tools do not, for example, the choice of cryptographic parameters can affect the accuracy of the output of a machine learning. It is essential for the success of FHE that developers, data scientists and other users have the resources to help them build this ecosystem safely and with FHE. (See the documentation for more details.)

Of course, achieving this will only be possible, reaching the stage where this knowledge can be applied requires a massive jump in the speed at which FHE software can be executed, and that is the other focus of this article. At Optalysys, we are designing hardware to accelerate the processing of the primitive operations in FHE (specifically in Radix TFHE), and we have ours first set of proto-type accelerators in the future.

The key to cracking this problem has to the same mathematical function that our technology has originally designed to compute: The Fourier transform. The mathematical tools and Fourier tools are very technical disciplines, but it is especially important for FHE.

It's the heart of calculations in FHE, it's the tool to multiply very large polynomials together, a task which is considerably easier when processed in Fourier space. This is both well known and well understood, and consequently it forms the cornerstone of most hardware designs. Already use Fourier transformation algorithms to make these multiplications easier.

However, the Fourier algorithm comes at the cost of many repeated operations that promise to be an entirely in parallel when implemented in hardware, but which are very costly if implemented in software. Consider that the calculation of the output of FHE, up to 80% of all the computational time is consumed by Fourier transformation calculations, and that number is expected to grow.

We take a different approach: we manipulate the properties of optics to perform Fourier transforms at nearly the speed of light. Our technology is a highly scalable optical chiplet that can be integrated into existing digital server systems.

This unique approach to calculating the Fourier transform has come about at exactly the right moment to provide acceleration for FHE, and we are currently developing our second generation hardware platform, which will be ready for sale in the second half of 2022. Our first prototype of this generation of our systems will be at sampling one of the three pilot-ready customers in the digital sector.

High-level description

Generally speaking (at least with respect to FHE), there are two different ways to perform computations on encrypted data. The first, and most common, is to represent the encrypted data as a series of binary operations and then homomorphically evaluate the logic gates. Alternatively, we can ask to apply to a piece of information some arbitrary univariate function, and then evaluate it. Although there are some gates and operations on the data can then be made homomorphically. Two very good tools for this are the Google E-transpiler and the Concrete Boolean library.

The second approach, which we will follow here, is to work directly with integers or continuous numbers (to within some level of precision) and to perform operations on whole numbers at a time. Although this approach is less flexible than working with binary strings, this approach can be much faster, than attempting to do the same operation via homomorphic Boolean logic. In the extreme, we'll apply this approach to the task of computing the maximum and minimum values of an array of numbers using only additions and univariate functions. We will again use the Concrete library, which makes this task much easier.

Before heading into our implementation, we'll briefly explain our thinking with unencrypted examples. Given two numbers a and b:

Our objective is to find their maximum using only additions, or functions which take only one argument. Give possibility for finding the maximum value is the following:

1. Subtract a value c to the opposite of b: $c = -b$

2. Compute the sum $d = a + c$

3. Define the positive part of d: $d = d \text{ if } d > 0 \text{ else } d = 0$ (otherwise, then $d=0$)

4. The minimum of a and b: $a + e$. Here, we can see that is the maximum value.

We can perform a similar trick to find the minimum:

1. Define the opposite of a: $c = -a$ as before.
2. Compute the sum $d = b + c$ as before.
3. Define the negative part of d: $d = d \text{ if } d < 0 \text{ else } d = 0$ (otherwise.

4. The minimum of a and b: $a + e$. Here, we can see that is the minimum value.

This approach takes advantage of the fact that we can, in the above described scheme, in FHE, additions and multiplication by a constant (here, -1) are just cost free. The cost of a bootstrap is the sum of the pre-set cost (constant/parameter) because they do not depend on the input. In each of these two algorithms, there is thus only one "cost" operation requiring a bootstrap: taking the positive or negative part of d.

Since you can compute the minimum and maximum of two numbers, it is easy to compute the maximum and minimum values of a larger set by doing multiple comparisons of pairs of arrays. For a set of N elements, this requires N comparisons.

Homomorphic version

Let us now illustrate this on a larger set. Consider an array of numbers representing a discretized sine function:

In this figure, it is easy to see that the maximum is 1 and the minimum is -1.

The catch, or course, is that we can't send this array directly to the server, in the case of FHE it is that the server whole computations will perform, does not have access to the private data. Instead, we first encrypt the data before sending it to the server. Each element of the array is encrypted individually, and the order of the array is shuffled so that it is not a single number but this still will have its attention to the original data. On encryption, the array looks basically random:

The advantage of this is obvious: since the data looks random (in fact it doesn't just look random: it can be shown that, without the secret key, it is indeed indistinguishable from random), it can be safely sent without risk of the data being stolen. In the following, we will show how we can nonetheless recover the maximum and minimum in encrypted data, as schematically illustrated in the new Figure.

Here, the dashed lines represent plaintext data, none of these is available to the server. The server only has access to encrypted data, represented by the blue dots. What we will show is how, using programmable bootstrapping and the FHE operations, we can compute the max and min values, when decrypted by the client, give the maximum and minimum of the original array (purple dotted lines).

Computing the maximum or minimum of two encrypted values

Preliminaries

To run the code in this tutorial, you will need a fast compiler (instructions to install one can be found on the zama.ai/concrete-boolean-blog). We'll use the Concrete library, which is based on the Zama Boolean library and uses Zama's fast and secure FHE. Once you have installed the compiler, you will also need to add a new dependency to your Cargo.toml file, in the [dependencies] section of the Cargo.toml file, add:

```
concrete-boolean = { version = "0.2.1", features = [ "boolean", "shortint", "integer", "nightly-avx512" ] }
```

Concrete-boolean, the Boolean library, is the simplest of the Zama libraries, which exposes only one gate: concrete-boolean-cli. This places the cryptographic parameters into a pre-set, secure, default set, which makes it easier to use.

In the following, we will go through the code step by step, explaining as we go. A full version of the code is available in the Appendix.

In the following, we start by creating functions that use Concrete to cryptographically compute the same sequence of steps that we demonstrated on the unencrypted example data. We then take these functions and show how we can apply them in a manner that scales well, allowing us to find the maximum and minimum values in these arrays without needing to decrypt the information!

Max algorithm

The first thing we need to the function to compute the maximum of two encrypted values. We can't just compare the ciphertexts – since a plaintext should not leak any information, the ciphertext is random looking, there is no way we can know anything about the larger value without the decryption key. Instead, we will use Concrete to construct the sequence of additions, comparisons and subtractions need to find the maximum value. We will also need to create a new ciphertext which is a different encryption of the larger value:

```
fn compare_ct(ct0:&mut LWE, ct1:&mut LWE, ct2:&mut LWE, ct3:&mut LWE,
              bsk:&LweBootstrapKeyOwned<u64>,
              ks:&LweKeyswitchKeyOwned<u64>,
              buffer:&mut [u64]){

    // Difference between the two ciphers
    lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct1,buffer);

    // Ciphertext0 multiplied by sum of the difference to encrypted
    // the maximum and minimum of the two ciphertexts
    lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct0,buffer);

    // Keyswitch bootstrap to shift the difference to positive
    // to make sure ciphertext_sum = ciphertext0+ciphertext1
    // (if it is the first ciphertext in the pair)
    keyswitch_bootstrap_single(ct0,ct2,bsk,ks,buffer);

    // Change the key back to the original LWE
    lwe_ciphertext_cloned_assign(ct0,ct3);

    // Subtract the second from ciphertext 1
    lwe_ciphertext_sub_assume_mod_power_of_2_inplace(ct0,ct1,buffer);

    // Add it to make the binary phase
    keyswitch_bootstrap_single(ct0,ct1,bsk,ks,buffer);

    // Keyswitch
    lwe_ciphertext_cloned_assign(ct0,ct2);
}
```

Show the function code in Concrete-boolean

This function takes six arguments:

- ct0,ct1,ct2,ct3 are LWE ciphertexts encrypted with the LWE version of TFHE. They encrypt the two values we want to compare.
- bsk is a key switching key a public key (by 'public' we mean that this key can be safely shared) in it can reveal any information on the message of the ciphertext. It allows us to perform a gate bootstrap (a message and the key to a different ciphertext encrypting the same message with a different key. It is needed because the programmable bootstrap will generally change the encryption key. The key switching key is used to revert to the original key.
- ks is the keyswitching key, required to perform the programmable bootstrap. Like bsk, it is a public key.
- buffer is a buffer, it is a set of memory that is temporarily used by the function to store data. It is used to avoid the allocation and deallocation of memory, which can be a bottleneck. Strictly speaking, this parameter is not really needed, we could use the same encoder as that of LWE_2_2 in LWE_2. However, we prefer to include it as an explicit argument to give the function more flexibility, as choosing a different encoder may improve the accuracy in some cases.

The function takes as an input ct0 and ct1 and stores the difference between the two messages (1-0) then takes the opposite of LWE_2 and adds the result to LWE_2:

```
lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct0,buffer);
```

Show the code in Concrete-boolean

computes (an encrypted copy of) the difference between the two messages (1-0) then takes the opposite of LWE_2 and adds the result to LWE_2:

```
lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct1,buffer);
```

Show the code in Concrete-boolean

computes the positive part using programmable bootstrapping thanks to the bootstrapping key bsk, in doesn computing the positive part, it also changes the encryption key back to that of LWE_2. It stores it in LWE_2. This is stored the answer to the function, as the positive LWE_2 LWE_2 gives us an encryption of the result, but under a different key than the original.

```
keyswitch_bootstrap_single(ct0,ct2,bsk,ks,buffer);
```

Show the code in Concrete-boolean

then computes the sum of the result and LWE_2. The operation

```
lwe_ciphertext_cloned_assign(ct0,ct3);
```

Show the code in Concrete-boolean

adds an encryption of 0, which has the effect of re-centering the encoder.

Finally, the function returns LWE(ct2) containing either a ciphertext if all operations were successful, or a CryptoAPIError error if one of the operations failed. The function ends with:

```
return;
```

Show the code in Concrete-boolean

We, fortunately, the key switching key is not absolutely necessary: we could also choose the input and output keys of the bootstrapping operation to be the same, and thus not need to change the key ciphertext. In this article, we choose the input and output keys independently to illustrate a very useful property in Concrete, since the runtime overhead of the key switching is quite small in practice.

Min algorithm

The function to compute the minimum of two values is very similar, and proceeds along the same lines as the maximum function:

```
fn compare_ct(ct0:&mut LWE, ct1:&mut LWE, ct2:&mut LWE, ct3:&mut LWE,
              bsk:&LweBootstrapKeyOwned<u64>,
              ks:&LweKeyswitchKeyOwned<u64>,
              buffer:&mut [u64]){

    // Difference between the two ciphers
    lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct1,buffer);

    // Ciphertext0 multiplied by sum of the difference to encrypted
    // the maximum and minimum of the two ciphertexts
    lwe_ciphertext_add_assume_mod_power_of_2_inplace(ct0,ct0,buffer);

    // Keyswitch bootstrap to shift the difference to positive
    // to make sure ciphertext_sum = ciphertext0+ciphertext1
    // (if it is the first ciphertext in the pair)
    keyswitch_bootstrap_single(ct0,ct2,bsk,ks,buffer);

    // Change the key back to the original LWE
    lwe_ciphertext_cloned_assign(ct0,ct3);

    // Subtract the second from ciphertext 1
    lwe_ciphertext_sub_assume_mod_power_of_2_inplace(ct0,ct1,buffer);

    // Add it to make the binary phase
    keyswitch_bootstrap_single(ct0,ct1,bsk,ks,buffer);

    // Keyswitch
    lwe_ciphertext_cloned_assign(ct0,ct2);
}
```

Show the code in Concrete-boolean

The only differences between this and the maximum array iterator are that we don't need to take the opposite of 2 and, of course, that we are using min(a,b) instead of max(a,b) in the last step.

Example

Let us show how to use these functions on a simple example. Create a file src/main.rs containing:

```
use concrete_boolean::prelude::*;

fn main() -> Result, Vec<bool>, Vec<bool>{
```

Show the code in Concrete-boolean

The code above is the start of the section should go in the main function.

We first define the encoder. We will work with values between 0 and 5 with 4 bits of accuracy. We set the number of bits of padding to 1. (See the concrete documentation for more information on the encoders, keys, and parameters.)

```
    // encoder
    // 4 bits
    // message 2 bits
    // carry 1 bit
    // padding 1 bit
    // range of values: 0..5
    // range of the carry: 0..1
    let encoder = FheUint2::encode(1_u8);
```

Show the code in Concrete-boolean

We then define the key-switching key, needed to convert a ciphertext to the key sk_lwe, to a ciphertext for sk_ch, and the bootstrapping key. They both take two parameters, which we set to 1. (See the Concrete documentation.)

```
    // define the key-switching key and bootstrapping key
    let (cks, sks) = gen_keys();
```

Show the code in Concrete-boolean

We now define a plaintext array of data, with 100 elements of security:

```
    // initial input
    // 100 elements
    let initialized_vec: Vec;
    for _ in 0..100{
        let mut ct = FheUint2::encrypt(0_u8,&cks);
        ct.1 = FheBool::encrypt(false,&cks);
        ct.0 = FheBool::encrypt(true,&cks);
        initialized_vec.push(ct);
    }
    let mut output_vec = initialized_vec.clone();
```

Show the code in Concrete-boolean

We then define the key-switching key, needed to convert a ciphertext to the key sk_lwe, to a ciphertext for sk_ch, and the bootstrapping key. They both take two parameters, which we set to 1. (See the Concrete documentation.)

```
    // define the key-switching key and bootstrapping key
    let (cks, sks) = gen_keys();
```

Show the code in Concrete-boolean

```
    //
    // base log: 2^5
    // number of levels: 2^5
    let ksk = LWEKSK::new(&sk_out, &sk_in, 5, 5);
    let bsk = LWEKSK::new(&sk_out, &sk_in, 5, 5);
```

main4.rs hosted with ❤ by GitHub view raw

That's all for the set-up. We now define the vector of messages:

```
    // messages
    let messages: Vec<f64> = vec![0.3, 0.5, 0.3, 0.6, 0.1, 0.3];
```

main5.rs hosted with ❤ by GitHub view raw

encrypt it:

```
    // encrypt the messages
    let ciphers: Vec<LWE>
        = messages.iter().map(|m| LWE::encode_encrypt(&sk_in, *m, &encoder))
                 .collect::<Result<Vec<LWE>, CryptoAPIError>>()?;
```

main6.rs hosted with ❤ by GitHub view raw

compute the maximum and minimum values:

```
    let cipher_max = compute_max_array(&ciphers, &ksk, &bsk, &encoder, &zero)?;
    let cipher_min = compute_min_array(&ciphers, &ksk, &bsk, &encoder, &zero)?;
```

main7.rs hosted with ❤ by GitHub view raw

decrypt the result:

```
    let mut max_val = cipher_max.decrypt_decode(&sk_in)?;
    let mut min_val = cipher_min.decrypt_decode(&sk_in)?;
```

main8.rs hosted with ❤ by GitHub view raw

round the values:

```
    max_val = (10.*max_val).round() / 10.;
    min_val = (10.*min_val).round() / 10.;
```

main9.rs hosted with ❤ by GitHub view raw

print the results and return `Ok(())`:

```
    println!("maximum: {}", max_val);
    println!("minimum: {}", min_val);

    Ok(())
```

main10.rs hosted with ❤ by GitHub view raw

Running the program using `cargo run --release`, you should get:

```
maximum: 0.6
minimum: 0.1
```

output_1.txt hosted with ❤ by GitHub view raw

Sort algorithm

While finding the minimum and maximum of a dataset can already be useful, in practice this truly shines when used as a building block for other algorithms, *e.g.*, sorting an array. In this section, we now show how to implement a basic sorting algorithm working entirely on encrypted data.

Function to compute the maximum and minimum of two numbers

The functions `compute_min` and `compute_max` written above are fairly fast (by FHE standards) as they require only one bootstrap each. However, they have two drawbacks which we want to avoid for a full sort function:

- They use the function `add_centered_inplace`, which is not officially documented.
- The encoder correction obtained by adding an encryption of 0 is not always perfect, and may require fine-tuning the offset of the encryption of 0.

Here we show a different version using the programmable bootstrap to reset the encoder. To minimise the number of such bootstrap operations, the same function returns both the minimum and the maximum of two numbers.

```
// compute the maximum and minimum of two encrypted numbers
//
// Arguments:
//
//  cipher_1: the first cipher
//  cipher_2: the second cipher
//  ksk: the key-changing key needed to revert the key change induced by the bootstrap
//  bsk: the boostrapping key
//  encoder: the encoder to use in the bootstrap
fn compute_max_min(cipher_1: &LWE, cipher_2: &LWE, ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder)
    -> Result<(LWE, LWE), CryptoAPIError>
{

    // difference between the two ciphers
    let cipher_diff = cipher_2.sub_with_padding(&cipher_1)?;

    // programmable bootstrap to check if the difference is positive
    let mut cipher_diff_pos = cipher_diff.bootstrap_with_function(bsk,
                                                        |x| if x >= 0. { x } else { 0. },
                                                        encoder)?;

    // change the key back to the original one
    cipher_diff_pos = cipher_diff_pos.keyswitch(ksk)?;

    // add the result to cipher_1
    let mut result_max = cipher_1.add_with_padding(&cipher_diff_pos)?;

    // subtract the result from cipher_2
    let mut result_min = cipher_2.sub_with_padding(&cipher_diff_pos)?;

    // reset the encoder
    result_max = result_max.bootstrap_with_function(bsk, |x| x, encoder)?;
    result_min = result_min.bootstrap_with_function(bsk, |x| x, encoder)?;
    result_max = result_max.keyswitch(ksk)?;
    result_min = result_min.keyswitch(ksk)?;

    Ok((result_max, result_min))
}
```

min_and_max.rs hosted with ❤ by GitHub view raw

This function will be about three times slower than the previous ones, but it eliminates the drift in the encoder.

Sorting algorithm

Once we can find the minimum of an array of numbers, one can sort it by successively computing the minima of the remaining elements. This requires *n(n-1) / 2* comparisons, where *n* is the array size. There are much faster sorting algorithms: for instance, the merge sort algorithm requires only $n \lceil \log n \rceil / 2$ comparisons in the worst case (where log denotes the logarithm in base 2). However, these algorithms involve conditionals which are not straightforward to evaluate homomorphically (for instance, merge sort requires taking one element from each of two arrays to merge and determine which is the largest, a process which can not be directly implemented without leaking information on the data). For this reason, in this article we will stick to a naive algorithm which, while slower, has the advantage of being easy to implement in FHE.

```
pub fn sort_array(ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder)
    -> Result<Vec<LWE>, Box<dyn std::error::Error>>
{
    // if ciphers contains less than two elements, just return it
    if ciphers.len() < 2 {
        return Ok(ciphers.to_vec())
    }

    // vector containing the result
    let mut results = ciphers.to_vec();

    for i in 0..(ciphers.len()-1) {
        for j in (i+1)..ciphers.len() {
            let (c_max, c_min) = compute_max_min(&results[i], &results[j], ksk, bsk, encoder)?;
            results[i] = c_min;
            results[j] = c_max;
        }
    }

    Ok(results)
}
```

sort_array.rs hosted with ❤ by GitHub view raw

As described above, this algorithm works iteratively: it first finds the minimum value of the array, then the minimum value of the other elements, and so on. To use it, replace the last line in the `main` function (`Ok(())`) with

```
// sort the array
let ciphers_sorted = sort_array(&ciphers, &ksk, &bsk, &encoder)?;

// decrypt the results
let mut decrypted
    = ciphers_sorted.iter().map(|c| { c.decrypt_decode_round(&sk_in) })
                 .collect::<Result<Vec<f64>, CryptoAPIError>>()?;

// round the results
for i in 0..decrypted.len() {
    decrypted[i] = (decrypted[i] / precision).round() * precision;
}

// print the result
println!("{:?}", decrypted);

Ok(())
```

main_bis.rs hosted with ❤ by GitHub view raw

Optical acceleration

As described in the introduction, the main problem currently facing FHE is speed: the security it provides comes with a huge runtime penalty of between 1,000× to 1,000,000× depending on the kind of computations being performed. This is way too high for most practical applications, so realising the full potential of FHE will require bringing these numbers down to below 10×.

Fortunately, progress in this domain is coming fast. On the algorithmic and software side, newer schemes like TFHE and efficient implementations like Concrete bring orders-of-magnitude speed-ups over earlier frameworks. Taking a slightly different direction, levelled homomorphic encryption can significantly reduce the runtime for evaluating boolean circuits with a pre-determined, small or medium depth.

On the hardware side, the US Defense Advanced research Project Agency (DARPA) has launched a project to explore hardware acceleration of lattice cryptography in general, and FHE in particular. Promising ideas involve large processor caches and registers to work on large integers, as well as dedicated single instruction multiple data (SIMD) instructions.

At Optalysys, we are following a different yet complementary approach: we are developing a silicon-photonic chip to compute the Fourier transform, the main bottleneck in FHE operations, faster and more efficiently than is possible on electronic hardware. Our system is modular and can integrate with both cutting-edge FHE software and hardware to compound the acceleration that each approach brings. We have previously described our technology in these fourarticles (and take a look at our website for even more documentation). Moreover, we can give an estimate for how quickly our system could tackle the problem at hand.

To this end, we have split the above code into two parts: a client side which encrypts and decrypts data, and a server side performing computations on encrypted data, running on different processes with TCP communication between them. We have also modified the server side to make use of a simulator for our upcoming optical chip. The code is not publicly released yet, but can be made available to participants on our hardware beta program. We have run the same problem (sorting an array of 10 values) on an Intel i7–9700K CPU @ 3.60GHz and on our optical simulator and measured the runtime. To make the comparison valid, we subtracted the runtime for key generation, encryption, decryption, and client-server communication.

When running electronically, and after subtracting some overhead, sorting the array took a bit more than 6s. By comparison, the optical simulator indicated the same operation will run in 0.05s on our silicon-optical chip. Our technology can thus accelerate this sequence of FHE operations by two orders of magnitude. Moreover, this speed-up can be compounded with software improvements and other forms of hardware acceleration, each of which could provide further order-of-magnitude improvements in the coming years.

Based on these results, we strongly believe that FHE will become a commercial reality and revolutionise how the world exchanges data in the next few years — and we are ready to be an integral part of this adventure.

Appendix: Full code

For completeness, we show here the full `Cargo.toml`, `lib.rs` and `main.rs` files for this tutorial. We create the new project using `cargo new --lib homomorphic_min_max`, compile it with `cargo build --release`, and run the example using `cargo run --release`.

`Cargo.toml`:

```
[package]
name = "homomorphic_min_max"
version = "0.1.0"
edition = "2021"

[dependencies]
concrete = "0.1.11"
```

Cargo.toml hosted with ❤ by GitHub view raw

`src/lib.rs`:

```
pub use concrete::*;


// compute the maximum of two encrypted numbers
//
// Arguments:
//
//  cipher_1: the first cipher
//  cipher_2: the second cipher
//  ksk: the key-changing key needed to revert the key change induced by the bootstrap
//  bsk: the boostrapping key
//  encoder: the encoder to use in the bootstrap
//  zero: an encryption of 0 with inverted encoder with respect to cipher_1 and cipher_2
fn compute_max(cipher_1: &LWE, cipher_2: &LWE, ksk: &LWEKSK, bsk: &LWEBSK,
               encoder: &Encoder, zero: &LWE)
    -> Result<LWE, CryptoAPIError>
{

    // difference between the two ciphers
    let cipher_diff = cipher_2.add_centered(&cipher_1.opposite()?)?;

    // programmable bootstrap to check if the difference is positive
    let mut cipher_diff_pos = cipher_diff.bootstrap_with_function(bsk,
                                                        |x| if x >= 0. { x } else { 0. },
                                                        encoder)?;

    // change the key back to the original one
    cipher_diff_pos = cipher_diff_pos.keyswitch(ksk)?;

    // add the result to cipher_1
    let mut result = cipher_1.add_centered(&cipher_diff_pos)?;

    // subtract 0 to reset the encoder offset
    result.add_centered_inplace(zero)?;

    Ok(result)
}


// compute the minimum of two encrypted numbers
//
// Arguments:
//
//  cipher_1: the first cipher
//  cipher_2: the second cipher
//  ksk: the key-changing key needed to revert the key change induced by the bootstrap
//  bsk: the boostrapping key
//  encoder: the encoder to use in the bootstrap
//  zero: an encryption of 0
fn compute_min(cipher_1: &LWE, cipher_2: &LWE, ksk: &LWEKSK, bsk: &LWEBSK,
               encoder: &Encoder, zero: &LWE)
    -> Result<LWE, CryptoAPIError>
{

    // difference between the two ciphers
    let cipher_diff = cipher_2.add_centered(&cipher_1.opposite()?)?;

    // programmable bootstrap to check if the difference is positive
    let mut cipher_diff_pos = cipher_diff.bootstrap_with_function(bsk,
                                                        |x| if x >= 0. { x } else { 0. },
                                                        encoder)?;

    // change the key back to the original one
    cipher_diff_pos = cipher_diff_pos.keyswitch(ksk)?;

    // subtract the result from cipher_2
    let mut result = cipher_2.add_centered(&cipher_diff_pos.opposite()?)?;

    // add 0 to reset the encoder offset
    result.add_centered_inplace(zero)?;

    Ok(result)
}


/// compute the maximum of an array of encrypted numbers
///
///  ciphers: an array of ciphertexts
///  ksk: the key-changing key needed to revert the key change induced by the bootstrap
///  bsk: the boostrapping key
///  encoder: the encoder to use in the bootstrap
///  zero: an encryption of 0 with inverted encoder with respect to cipher_1 and cipher_2
pub fn compute_max_array(ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder, zero: &LWE)
    -> Result<LWE, Box<dyn std::error::Error>>
{
    // compute the opposite of 0 to reverse the encoder
    let zero = zero.opposite()?;

    // convert the array of ciphers into an iterator
    let mut ciphers_iter = ciphers.iter();

    // set the result to the first cipher, or return an error if the iterator is empty
    let mut result = ciphers_iter.next().ok_or("Empty cipher array!".to_string())?.clone();

    // loop over the other ciphers
    for cipher in ciphers_iter {

        // get the maximum of the current result and next cipher
        result = compute_max(&result, cipher, ksk, bsk, encoder, &zero)?;
    }

    Ok(result)
}


/// compute the minimum of an array of encrypted numbers
///
///  ciphers: an array of ciphertexts
///  ksk: the key-changing key needed to revert the key change induced by the bootstrap
///  bsk: the boostrapping key
///  encoder: the encoder to use in the bootstrap
///  zero: an encryption of 0 with inverted encoder with respect to cipher_1 and cipher_2
pub fn compute_min_array(ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder, zero: &LWE)
    -> Result<LWE, Box<dyn std::error::Error>>
{

    // convert the array of ciphers into an iterator
    let mut ciphers_iter = ciphers.iter();

    // set the result to the first cipher, or return an error if the iterator is empty
    let mut result = ciphers_iter.next().ok_or("Empty cipher array!".to_string())?.clone();

    // loop over the other ciphers
    for cipher in ciphers_iter {

        // get the minimum of the current result and next cipher
        result = compute_min(&result, cipher, ksk, bsk, encoder, &zero)?;
    }

    Ok(result)
}


// compute the maximum and minimum of two encrypted numbers
//
// Arguments:
//
//  cipher_1: the first cipher
//  cipher_2: the second cipher
//  ksk: the key-changing key needed to revert the key change induced by the bootstrap
//  bsk: the boostrapping key
//  encoder: the encoder to use in the bootstrap
fn compute_max_min(cipher_1: &LWE, cipher_2: &LWE, ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder)
    -> Result<(LWE, LWE), CryptoAPIError>
{

    // difference between the two ciphers
    let cipher_diff = cipher_2.sub_with_padding(&cipher_1)?;

    // programmable bootstrap to check if the difference is positive
    let mut cipher_diff_pos = cipher_diff.bootstrap_with_function(bsk,
                                                        |x| if x >= 0. { x } else { 0. },
                                                        encoder)?;

    // change the key back to the original one
    cipher_diff_pos = cipher_diff_pos.keyswitch(ksk)?;

    // add the result to cipher_1
    let mut result_max = cipher_1.add_with_padding(&cipher_diff_pos)?;

    // subtract the result from cipher_2
    let mut result_min = cipher_2.sub_with_padding(&cipher_diff_pos)?;

    // reset the encoder
    result_max = result_max.bootstrap_with_function(bsk, |x| x, encoder)?;
    result_min = result_min.bootstrap_with_function(bsk, |x| x, encoder)?;
    result_max = result_max.keyswitch(ksk)?;
    result_min = result_min.keyswitch(ksk)?;

    Ok((result_max, result_min))
}


/// sort an array of ciphertexts
pub fn sort_array(ciphers: &[LWE], ksk: &LWEKSK, bsk: &LWEBSK, encoder: &Encoder)
    -> Result<Vec<LWE>, Box<dyn std::error::Error>>
{
    // if ciphers contains less than two elements, just return it
    if ciphers.len() < 2 {
        return Ok(ciphers.to_vec())
    }

    // vector containing the result
    let mut results = ciphers.to_vec();

    for i in 0..(ciphers.len()-1) {
        for j in (i+1)..ciphers.len() {
            let (c_max, c_min) = compute_max_min(&results[i], &results[j], ksk, bsk, encoder)?;
            results[i] = c_min;
            results[j] = c_max;
        }
    }

    Ok(results)

}
```

lib.rs hosted with ❤ by GitHub view raw

`src/main.rs`:

```
use homomorphic_min_max::*;

fn main() -> Result<(), Box<dyn std::error::Error>> {

    // encoder
    let encoder = Encoder::new(0., 1., 4, 2)?;

    // expected precision
    let precision = 0.2;

    // secret keys
    let sk_rlwe = RLWESecretKey::new(&RLWE128_1024_1);
    let sk_in = LWESecretKey::new(&LWE128_1024);
    let sk_out = sk_rlwe.to_lwe_secret_key();

    // encryption of 0
    let zero = LWE::encode_encrypt(&sk_in, 0., &encoder)?;

    // key switching key
    let ksk = LWEKSK::new(&sk_out, &sk_in, 5, 5);

    // bootstrapping key
    let bsk = LWEBSK::new(&sk_in, &sk_rlwe, 5, 5);

    // messages
    let messages: Vec<f64> = vec![0.2, 0.4, 0.2, 0.6, 0.2, 0.4];

    // encrypt the messages
    let ciphers: Vec<LWE>
        = messages.iter().map(|m| LWE::encode_encrypt(&sk_in, *m, &encoder))
                 .collect::<Result<Vec<LWE>, CryptoAPIError>>()?;

    // perform the calculation
    let cipher_max = compute_max_array(&ciphers, &ksk, &bsk, &encoder, &zero)?;
    let cipher_min = compute_min_array(&ciphers, &ksk, &bsk, &encoder, &zero)?;

    // decrypt
    let mut max_val = cipher_max.decrypt_decode(&sk_in)?;
    let mut min_val = cipher_min.decrypt_decode(&sk_in)?;

    // round the results
    max_val = (max_val / precision).round() * precision;
    min_val = (min_val / precision).round() * precision;

    // print the result
    println!("maximum: {}", max_val);
    println!("minimum: {}", min_val);

    // sort the array
    let ciphers_sorted = sort_array(&ciphers, &ksk, &bsk, &encoder)?;

    // decrypt the results
    let mut decrypted
        = ciphers_sorted.iter().map(|c| { c.decrypt_decode_round(&sk_in) })
                     .collect::<Result<Vec<f64>, CryptoAPIError>>()?;

    // round the results
    for i in 0..decrypted.len() {
        decrypted[i] = (decrypted[i] / precision).round() * precision;
    }

    // print the result
    println!("{:?}", decrypted);

    Ok(())
}
```

main.rs hosted with ❤ by GitHub view raw

Optalysys

British Assessment Bureau ISO 9001 QUALITY MANAGEMENT

Telephone: +44 (0) 1977 551615 Email: info@optalysys.com

READ THE WHITE PAPER DOCUMENTS PRIVACY POLICY CONTACT US