nufhe Documentation

Release 0.0.2

Bogdan Opanchuk

Feb 14, 2019

Contents

CHAPTER 1

Contents

1.1 Implementation details

1.1.1 Polynomial multiplication

The main bottleneck of `NuFHE` gates is bootstrapping, and in it the most time is taken by multiplication of polynomials. In the FHE scheme used, the polynomials are negacyclic (modulo $x^N + 1$, where $N = 1024$ by default), defined for integers modulo 2^{32} (with the coefficients stored as signed 32-bit integers). One of the factors always has coefficients in range $[-1024, 1024)$. Two methods are can be used for multiplication, convolution via FFT and convolution via NTT (number theory transform)[1].

FFT

Since the polynomials are negacyclic, it is not enough to transform the coefficients to Fourier space, multiply them and then transform the result back — that would correspond to the regular cyclic convolution, that is multiplication of polynomials modulo $x^N - 1$. Our polynomials are negacyclic, which makes things slightly more complicated.

A straightforward approach is to extend the array of each polynomial's coefficients, turning $(a_0, \ldots, a_{N-1})$ into $(a_0, \ldots, a_{N-1}, -a_0, \ldots, -a_{N-1})$. This way the regular convolution of these extended arrays will result in the negacyclic convolution of original arrays.

The Fourier transform of such a signal (of total size $2N$) results in an array containing only N non-zero elements (in the positions with odd indices), of which the last $N/2$ are complex conjugates of the first $N/2$. This indicates that a transform of size $N/2$ should be sufficient to obtain it. And indeed, if one uses standard approaches[2] and takes advantage of the two properties of the extended array: real elements and antiperiodicity, the problem can be reduced to a transform of size $N/2$ with some pre- and post-processing.

The algorithm built this way maps poorly on the execution model of a GPU, because the pre- and post-processing

[1] *J. M. Pollard,* "The Fast Fourier Transform in a Finite Field", Mathematics of Computation 25(114), 365–365 (1971).

[2] *L. R. Rabiner,* "On the Use of Symmetry in FFT Computation", IEEE Transactions on Acoustics Speech and Signal Processing 27(3), 233–239 (1979).

required is not perfectly parallelizable. In `NuFHE` a technique based on D. J. Bernstein's tangent FFT[3][4] is used, which in its core still has an $N/2$-size Fourier transform, but with a much simpler processing.

The algorithm is as follows. Given a vector $\boldsymbol{a}$ of length N, we define the forward transform as:

$$\boldsymbol{c} = \mathrm{TFFT}\left[\boldsymbol{a}\right] = \mathrm{FFT}\left[\boldsymbol{b}\right],$$

where $\boldsymbol{b}$ is a $N/2$-vector with the elements

$$b_j = \left(a_j - i a_{j+N/2}\right) w^j, \quad j \in [0, N/2),$$

and $w = \exp\left(-\pi i/N\right)$ is a $2N$-th root of unity. Note that the complex vector $\boldsymbol{c}$ consists of the first $N/2$ non-zero elements Fourier-transformed extended coefficient array described above, except in a different order. Since we will only use the Fourier-space array for convolution, the order does not matter.

The inverse transform $\boldsymbol{a} = \mathrm{ITFFT}\left[\boldsymbol{c}\right]$ is calculated as:

$$\boldsymbol{b} = \left(\mathrm{IFFT}\left[\boldsymbol{c}\right]\right)^*$$

$$a_j = \mathrm{Re}\left(b_j w^j\right), \quad a_{j+N/2} = \mathrm{Im}\left(b_j w^j\right), \quad j \in [0, N/2).$$

Using this pair of transforms, the negacyclic multiplication of two polynomials with coefficients $\boldsymbol{u}$ and $\boldsymbol{v}$ is performed simply as

$$\mathrm{ITFFT}\left[\mathrm{TFFT}\left[\boldsymbol{u}\right] \circ \mathrm{TFFT}\left[\boldsymbol{v}\right]\right],$$

where $\circ$ stands for elementwise multiplication of two vectors.

Such pre- and post-processing is simple, perfectly parallel and requires only sequential memory access, which makes it ideal for use on a GPU.

Note that this method will only work as long as the maximum possible result does not exceed the capacity of the floating point number used for the underlying FFT (since the modulo 2^{32} can only be taken after the FFT). In our case we have coefficients limited by 32 bits and 11 bits respectively, plus 10 bits due to the polynomial size (1024), which fits into 53 bits of the double-precision floating-point significand.

NTT

Alternatively, polynomial multiplication can be performed using an NTT, which is essentially an FFT operating on the elements of a finite field of size M, where M is a prime number. Same as in the case of FFT, using an unmodified pair NTT-INTT results in the regular cyclic convolution, and additional steps are necessary to turn it into the negacyclic one. The scheme is very similar to the one used for FFT, and is described in[5].

Given a vector $\boldsymbol{a}$ of length N, we define the forward transform as:

$$\boldsymbol{c} = \mathrm{TNTT}\left[\boldsymbol{a}\right] = \mathrm{NTT}\left[\boldsymbol{b}\right],$$

where $\boldsymbol{b}$ is a N-vector with the elements

$$b_j = a_j w^j, \quad j \in [0, N),$$

$w = g^{(M-1)/(2N)}$, and g is a primitive element of the field. This means that w is a $2N$-th root of unity in the field, just like the one in the FFT section. Note that $M - 1$ must be a multiple of $2N$.

[3] *D. J. Bernstein,* "The Tangent FFT", Applied Algebra, Algebraic Algorithms and Error-Correcting Codes 291–300 (2007).
[4] *D. J. Bernstein,* "Fast multiplication and its applications", Algorithmic Number Theory 44 (2008).
[5] *P. Longa* and *M. Naehrig,* "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography".

Correspondingly, the inverse transform $\boldsymbol{a} = \mathrm{ITNTT}\left[\boldsymbol{c}\right]$ is

$$\boldsymbol{a} = \mathrm{INTT}\left[\boldsymbol{c}\right]$$

$$a_j = b_j w^{-j}, \quad j \in [0, N).$$

Same as in the case of FFT, the negacyclic multiplication of two polynomials with coefficients $\boldsymbol{u}$ and $\boldsymbol{v}$ is performed as

$$\mathrm{ITNTT}\left[\mathrm{TNTT}\left[\boldsymbol{u}\right] \circ \mathrm{TNTT}\left[\boldsymbol{v}\right]\right].$$

Since the polynomial coefficients are signed integers, they have to be converted to the field elements first, by taking them modulo M. The field must be large enough to accommodate the full range of possible outcome values (53 bits by deafult), before modulo 2^{32} can be taken.

The choice of modulus in NTT

`NuFHE`, following cuFHE, uses a specifically chosen modulus and root of unity, which allow for some performance optimizations.

The modulus (the size of the finite field) is chosen to be $M = 2^{64} - 2^{32} + 1$. It has several important properties. First, since the field elements are stored in 64-bit unsigned integers, arithmetic operations using this modulus can take advantage of its form. For example, $a \bmod M$ is simply a if $a < M$ and $a + \mathrm{UInt32}(-1)$ if $a \geq M$. Similar optimizations can be employed for subtraction, multiplication or bitshift.

Second, $M - 1$ is a multiple of 2^{32}, which means that it supports NTTs up to that size (when the size is a power of 2), and multiplication of polynomials of up to the size 2^{31}.

The N-th root of unity w_N used in NTT can theoretically be based on any primitive element g by setting $w = g^{(M-1)/N}$. `NuFHE` (again, following `cuFHE`) uses a "magic" constant $c = 12037493425763644479$, which is a $((M-1)/2^{32})$-th power of some primitive element. Therefore, for a given N (which must be a power of 2), one takes $w_N = c^{2^{32}/N}$. The advantage of using this constant is that $c^{2^{32}/64} = 8$, which means that in NTT one can replace most of multiplications by various powers of w_N by modulo bitshifts, which are much faster.

1.1.2 References

1.2 API reference

1.2.1 High-level api

class `nufhe.`**`Context`**(*rng=None, device_id=None, api=None, interactive=False, include_devices=None, exclude_devices=None, include_platforms=None, exclude_platforms=None*)

An object encapsulating an execution environment on a GPU.

Parameters

- **`rng`** – a random number generator which will be used wherever randomness is required. Can be an instance of one of the *Random number generators* (`DeterministicRNG` by default).
- **`device_id`** (`Optional[DeviceID]`) – if provided, uses this GPGPU device (and API) to create the context.

- **interactive** – if True, an interactive dialogue will be shown allowing one to choose the GPGPU device to use. If False, the first device satisfying the filters (see below) will be chosen.
- **api** –
- **include_devices** –
- **exclude_devices** –
- **include_platforms** –
- **exclude_platforms** – see *find_devices()*.

decrypt (*secret_key*, *ciphertext*)
Decrypts a message.

The low-level analogue: *decrypt()*.

Returns a numpy.ndarray object of the type numpy.bool and the same *shape* as ciphertext.

encrypt (*secret_key*, *message*)
Encrypts a message (a list or a numpy array treated as an array of booleans).

The low-level analogue: *encrypt()*.

Returns an *LweSampleArray* object with the same *shape* as the given array.

load_ciphertext (*file_or_bytestring*)
Load a ciphertext (a *LweSampleArray* object) serialized with *LweSampleArray.dump()* or *LweSampleArray.dumps()* into the context memory space.

The low-level analogues: *LweSampleArray.load()* and *LweSampleArray.loads()*.

Returns an *LweSampleArray* object

load_cloud_key (*file_or_bytestring*)
Load a secret key (a *NuFHECloudKey* object) serialized with *NuFHECloudKey.dump()* or *NuFHECloudKey.dumps()* into the context memory space.

The low-level analogues: *NuFHECloudKey.load()* and *NuFHECloudKey.loads()*.

Returns a *NuFHECloudKey* object

load_secret_key (*file_or_bytestring*)
Load a secret key (a *NuFHESecretKey* object) serialized with *NuFHESecretKey.dump()* or *NuFHESecretKey.dumps()* into the context memory space.

The low-level analogues: *NuFHESecretKey.load()* and *NuFHESecretKey.loads()*.

Returns a *NuFHESecretKey* object

make_cloud_key (*secret_key*)
Creates a cloud key matching the given secret key.

The low-level analogue: *NuFHECloudKey.from_rng()*.

Returns a *NuFHECloudKey* object.

make_key_pair (***params*)
Creates a pair of a secret key and a matching cloud key.

The low-level analogue: *make_key_pair()*.

Returns a tuple of a *NuFHESecretKey* and a *NuFHECloudKey* objects.

make_secret_key(***params*)

Creates a secret key, with `params` used to initialize a *NuFHEParameters* object.

The low-level analogue: *NuFHESecretKey.from_rng()*.

Returns a *NuFHESecretKey* object.

make_virtual_machine(*cloud_key, perf_params=None*)

Creates an FHE "virtual machine" which can execute logical gates using the given cloud key. Optionally, one can pass a *PerformanceParameters* object which will be specialized for the GPU device of the context and used in all the gate calls.

Returns a *VirtualMachine* object.

nufhe.**find_devices**(*api=None, include_devices=None, exclude_devices=None, include_platforms=None, exclude_platforms=None*)

Returns a list of computation device identifiers for the given API and selection criteria. If there are several platforms with suitable devices, only the first one will be used (so if you need a specific platform, use the corresponding masks).

Parameters

- **api** – the GPGPU backend to use, one of `None`, `"CUDA"` and `"OpenCL"`. If `None` is given, an arbitrary available backend will be chosen.
- **include_devices** – a list of strings; only devices with one of the strings present in the name will be included.
- **exclude_devices** – a list of strings; devices with one of the strings present in the name will be excluded.
- **include_platforms** – a list of strings; only platforms with one of the strings present in the name will be included.
- **exclude_platforms** – a list of strings; platforms with one of the strings present in the name will be excluded.

Returns a list of *DeviceID* objects.

class nufhe.api_high_level.**DeviceID**(*api_id, platform_id, device_id*)

An identifier of a computation device suitable to run NuFHE. Obtained from *find_devices()*. Can be passed to another thread/process and used to create a *Context* object.

api_name

The name of the API (`"CUDA"` or `"OpenCL"`).

platform_name

The name of the platform.

device_name

The name of the device.

class nufhe.api_high_level.**VirtualMachine**(*thread, cloud_key, perf_params=None*)

A fully encrypted virtual machine capable of executing gates on ciphertexts (*LweSampleArray* objects) using an encapsulated cloud key.

gate_<operator>(*args, dest: LweSampleArray=None)

Calls one of *Logical gates*, using the context, the cloud key, and the performance parameters of the virtual machine.

If `dest` is `None`, creates a new ciphertext and uses it to store the output of the gate; otherwise `dest` is used for that purpose.

Returns an *LweSampleArray* object with the result of the gate application.

empty_ciphertext (*shape*)
Returns an unitialized ciphertext (an *LweSampleArray* object).

The low-level analogue: *empty_ciphertext()*.

load_ciphertext (*file*)
Load a ciphertext (a *LweSampleArray* object) serialized with *LweSampleArray.dump* into the context memory space.

The low-level analogue: *LweSampleArray.load*.

Returns an *LweSampleArray* object

1.2.2 Parameters and keys

class nufhe.**NuFHEParameters** (*transform_type='NTT'*, *tlwe_mask_size=1*)
Parameters of the FHE scheme.

Parameters transform_type – 'NTT' or 'FFT', specifying the transform to be used for internal purposes. 'FFT' is generally faster, but may not be supported on some videocards (since it requires double precision floating point numbers).

Note: The default parameters correspond to about 128 bits of security.

class nufhe.**NuFHESecretKey** (*params*, *lwe_key*)
A secret key for the FHE scheme.

params
A *NuFHEParameters* object.

dump (*file_obj*)
Serialize into the given `file_obj`, a writeable file-like object.

dumps ()
Serialize into a bytestring.

classmethod from_rng (*thr*, *params*, *rng*)
Generate a new secret key.

Parameters

- **thr** – a `reikna Thread` object.
- **params** (`NuFHEParameters`) – FHE scheme parameters.
- **rng** – an RNG object, one of *Random number generators*.

classmethod load (*file_obj*, *thr*)
Deserialize from the given `file_obj`, a readable file-like object, using the `reikna` thread `thr` to store arrays.

classmethod loads (*s*, *thr*)
Deserialize from the given bytestring using the `reikna` thread `thr` to store arrays.

class nufhe.**NuFHECloudKey** (*params*, *bootstrap_key*, *keyswitch_key*)
A cloud key for the FHE scheme.

params
A *NuFHEParameters* object.

dump (*file_obj*)
Serialize into the given `file_obj`, a writeable file-like object.

dumps ()
Serialize into a bytestring.

classmethod from_rng (*thr*, *params*, *rng*, *secret_key*, *perf_params=None*)
Generate a new cloud key based on the given secret key.

Parameters

- **thr** – a `reikna` `Thread` object.
- **params** (`NuFHEParameters`) – FHE scheme parameters.
- **rng** – an RNG object, one of *Random number generators*.
- **secret_key** (`NuFHESecretKey`) – the secret key object.
- **perf_params** (`Optional[PerformanceParametersForDevice]`) – an override for performance parameters.

classmethod load (*file_obj*, *thr*)
Deserialize from the given `file_obj`, a readable file-like object, using the `reikna` thread `thr` to store arrays.

classmethod loads (*s*, *thr*)
Deserialize from the given bytestring using the `reikna` thread `thr` to store arrays.

nufhe.**make_key_pair** (*thr*, *rng*, ***params*)
Creates a pair of `NuFHESecretKey` and `NuFHECloudKey` corresponding to `NuFHEParameters` created with keywords `params`.

1.2.3 Random number generators

class nufhe.**DeterministicRNG** (*seed=None*)
A fast, but not cryptographically secure RNG. Useful for testing, since it allows seeding the initial state.

class nufhe.**SecureRNG**
A cryptographically secure RNG provided by the OS.

Note: This RNG can be very slow, leading to cloud key creation times of the order of minutes. Encryption is not affected too much (the required amount of random numbers is much lower).

1.2.4 Encryption/decryption

nufhe.**encrypt** (*thr*, *rng*, *key*, *message*)
Encrypts a message.

Parameters

- **rng** – an RNG object, one of *Random number generators*.
- **key** (`NuFHESecretKey`) – the secret key.
- **message** – a `numpy` array of bit values to encrypt; if the `dtype` is not `numpy.bool`, it will be converted to `numpy.bool`.

Returns an `LweSampleArray` object with the same *shape* as the given array.

`nufhe.`**`decrypt`**(*thr*, *key*, *ciphertext*)

Decrypts a message.

Parameters

- **`key`** (`NuFHESecretKey`) – the secret key.
- **`ciphertext`** (`LweSampleArray`) – an encrypted message.

Returns a `numpy.ndarray` object of the type `numpy.bool` and the same *shape* as `ciphertext`.

`nufhe.`**`empty_ciphertext`**(*thr*, *params*, *shape*)

Creates an uninitialized *`LweSampleArray`* with the shape `shape`.

`class` `nufhe.`**`LweSampleArray`**(*params*, *a*, *b*, *current_variances*)

A ciphertext object.

`shape`

The shape of the encrypted plaintext message.

`__getitem__`(*index*)

Returns a view over the ciphertext (still a *`LweSampleArray`* object). The indexing works in the same way as if it was a regular `numpy` array with the shape `shape`.

`dump`(*file_obj*)

Serialize into the given `file_obj`, a writeable file-like object.

`dumps`()

Serialize into a bytestring.

`classmethod load`(*file_obj*, *thr*)

Deserialize from the given `file_obj`, a readable file-like object, using the `reikna` thread `thr` to store arrays.

`classmethod loads`(*s*, *thr*)

Deserialize from the given bytestring using the `reikna` thread `thr` to store arrays.

1.2.5 Logical gates

Unary gates

`nufhe.`**`gate_constant`**(*thr*, *cloud_key*, *result*, *vals*, *perf_params=None*)

Fill each bit of the ciphertext `result` with the trivial encryption of the plaintext values from `vals` (which will be converted to `bool`).

`vals` should be an array or a list with a shape broadcastable to the shape of `result`, or a scalar value.

Note: "Trivial encryption" means that the result of this gate does not require a secret key for decryption, and cannot be used to implement public key encryption. Its intended purpose is to initialize constants in bootstrapped circuits.

Not bootstrapped; `perf_params` does not have any effect and is only present for the sake of API uniformity.

Parameters

- **`thr`** – a `reikna Thread` object.
- **`cloud_key`** (`NuFHECloudKey`) – the cloud key.

- **result** (LweSampleArray) – an empty ciphertext where the result will be stored. Must be the same shape as the vals array.
- **vals** – a numpy.bool array (or anything castable to it) used to fill the ciphertext.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_copy**(*thr*, *cloud_key*, *result*, *a*, *perf_params=None*)

Copy the contents of the ciphertext a to result.

Not bootstrapped; perf_params does not have any effect and is only present for the sake of API uniformity.

The shape of a should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the source ciphertext.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_not**(*thr*, *cloud_key*, *result*, *a*, *perf_params=None*)

Homomorphic NOT gate. Applied elementwise on an encrypted array of bits.

Not bootstrapped; perf_params does not have any effect and is only present for the sake of API uniformity.

The shape of a should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the source ciphertext.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

Binary gates

nufhe.**gate_and**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped AND gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.

- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_or**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped OR gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_xor**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped XOR gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_nand**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped NAND gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_nor**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped NOR gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a `reikna` `Thread` object.
- **cloud_key** (`NuFHECloudKey`) – the cloud key.
- **result** (`LweSampleArray`) – an empty ciphertext where the result will be stored.
- **a** (`LweSampleArray`) – the ciphertext with the first argument.
- **b** (`LweSampleArray`) – the ciphertext with the second argument.
- **perf_params** (`Optional[`*`PerformanceParametersForDevice`*`]`) – an override for performance parameters.

`nufhe.`**`gate_xnor`**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped XNOR gate. Applied elementwise on two encrypted arrays of bits.

The shapes of `a` and `b` should be broadcastable to the shape of `result`.

Parameters

- **thr** – a `reikna` `Thread` object.
- **cloud_key** (`NuFHECloudKey`) – the cloud key.
- **result** (`LweSampleArray`) – an empty ciphertext where the result will be stored.
- **a** (`LweSampleArray`) – the ciphertext with the first argument.
- **b** (`LweSampleArray`) – the ciphertext with the second argument.
- **perf_params** (`Optional[`*`PerformanceParametersForDevice`*`]`) – an override for performance parameters.

`nufhe.`**`gate_andny`**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped ANDNY (*(not a) and b*) gate. Applied elementwise on two encrypted arrays of bits.

The shapes of `a` and `b` should be broadcastable to the shape of `result`.

Parameters

- **thr** – a `reikna` `Thread` object.
- **cloud_key** (`NuFHECloudKey`) – the cloud key.
- **result** (`LweSampleArray`) – an empty ciphertext where the result will be stored.
- **a** (`LweSampleArray`) – the ciphertext with the first argument.
- **b** (`LweSampleArray`) – the ciphertext with the second argument.
- **perf_params** (`Optional[`*`PerformanceParametersForDevice`*`]`) – an override for performance parameters.

`nufhe.`**`gate_andyn`**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)

Homomorphic bootstrapped ANDYN (*a and (not b)*) gate. Applied elementwise on two encrypted arrays of bits.

The shapes of `a` and `b` should be broadcastable to the shape of `result`.

Parameters

- **thr** – a `reikna` `Thread` object.
- **cloud_key** (`NuFHECloudKey`) – the cloud key.
- **result** (`LweSampleArray`) – an empty ciphertext where the result will be stored.
- **a** (`LweSampleArray`) – the ciphertext with the first argument.

- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_orny**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)
Homomorphic bootstrapped ORNY (*(not a) or b*) gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

nufhe.**gate_oryn**(*thr*, *cloud_key*, *result*, *a*, *b*, *perf_params=None*)
Homomorphic bootstrapped ORYN (*a or (not b)*) gate. Applied elementwise on two encrypted arrays of bits.

The shapes of a and b should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

Ternary gates

nufhe.**gate_mux**(*thr*, *cloud_key*, *result*, *a*, *b*, *c*, *perf_params=None*)
Homomorphic bootstrapped MUX (*b if a else c*, or, equivalently, *(a and b) or ((not a) and c)*) gate. Applied elementwise on three encrypted arrays of bits.

The shapes of a, b and c should be broadcastable to the shape of result.

Parameters

- **thr** – a reikna Thread object.
- **cloud_key** (NuFHECloudKey) – the cloud key.
- **result** (LweSampleArray) – an empty ciphertext where the result will be stored.
- **a** (LweSampleArray) – the ciphertext with the first argument.
- **b** (LweSampleArray) – the ciphertext with the second argument.
- **c** (LweSampleArray) – the ciphertext with the third argument.

- **perf_params** (Optional[*PerformanceParametersForDevice*]) – an override for performance parameters.

1.2.6 Performance parameters

class nufhe.**PerformanceParameters**(*nufhe_params*, *ntt_base_method=None*, *ntt_mul_method=None*, *ntt_lsh_method=None*, *use_constant_memory_multi_iter=None*, *use_constant_memory_single_iter=None*, *transforms_per_block=None*, *single_kernel_bootstrap=None*, *low_end_device=None*)

Advanced performance settings for bootstrapping.

For all the optional parameters below, if None is given, the library will attempt to select the best performing variant, given the available information.

Parameters

- **nufhe_params** – a *NuFHEParameters* object.
- **ntt_base_method** – 'cuda_asm' or 'c'; An algorithm used in NTT for modulo addition, modulo subtraction, and modulus.
- **ntt_mul_method** – one of 'cuda_asm', 'c_from_asm' and 'c'; An algorithm used in NTT for modulo multiplication.
- **ntt_lsh_method** – one of 'cuda_asm', 'c_from_asm' and 'c'; An algorithm used in NTT for modulo bitshift.

Note: 'cuda_asm' is only available for CUDA backend. When available, it is usually the fastest variant, or close to it.

Parameters

- **use_constant_memory_multi_iter** – use constant GPU memory (as opposed to global memory) for precalculated coefficients in NTT/FFT in kernels where one of these transformations is executed multiple times per kernel call.
- **use_constant_memory_single_iter** – use constant GPU memory (as opposed to global memory) for precalculated coefficients in NTT/FFT in kernels where one of these transformations is executed once per kernel call.

Note: Using constant memory is usually beneficial on fast videocards if the transformation is executed many times per kernel call.

Parameters **transforms_per_block** – a positive integer value, denoting how many separate transforms to execute in parallel on a single GPU multiprocessor (compute unit).

Note: On most videocards 1 to 4 transforms is supported for NTT, and 1 to 8 for FFT. More transforms does not necessarily mean better performance, since parallel threads on the same compute unit compete for resources.

Since it is not trivial to determine the maximum in advance, if the requested number is greater than that, it will be dynamically reduced to the maximum possible value.

Parameters **single_kernel_bootstrap** – if `True`, execute bootstrap in a single kernel, instead of many separate kernel calls in a loop.

Note: Single kernel bootstrap is only supported for default FHE parameters, and needs the videocard to support a certain amount of parallel threads on a compute unit (256 for FFT, 512 for NTT). If available, it is usually significantly faster, partially due to lower kernel call overhead.

Parameters **low_end_device** – if `True`, the optimal values for low-end videocards will be picked. If `None`, the decision will be made based on the number of compute units the device has.

for_device(*device_params*)
Specialize performance parameters for the given device (using a Reikna `DeviceParams` object).

Returns a *PerformanceParametersForDevice* object.

class nufhe.performance.**PerformanceParametersForDevice**(*perf_params*, *device_params*)

1.2.7 Utility functions

nufhe.**clear_computation_cache**(*thr*)
Clear the cache of computation objects compiled for the given `reikna` thread `thr`. This will help ensure a correct realease of the thread's resources when the other references to it go out of scope (which is especially important for multi-threading applications using CUDA).

Note: *Context* objects call this function automatically on destruction.

1.3 Version history

1.3.1 0.0.2 (14 Feb 2019)

- **CHANGED:** a `PerformanceParameters` object needs to be specialized for the device used (by calling its `for_device()` method) before passing it to gates.
- **CHANGED:** instead of using `numpy.random.RandomState` for key generation and encryption, `DeterministicRNG` and `SecureRNG` are available instead. The former is the wrapped `RandomState`, fast, but not cryptographically secure; the latter is the secure random source provided by the OS, which can be rather slow.
- ADDED: a high-level API hiding the Reikna details and removing some boilerplate.
- ADDED: shape checks in gate functions that take into account possible broadcasting.
- ADDED: `dumps()` and `loads()` methods for `NuFHESecretKey`, `NuFHECloudKey` and `LweSampleArray` for serializing to/from bytestrings. The `Context`'s `load_secret_key`, `load_cloud_key` and `load_ciphertext` also take bytestrings as arguments.

- ADDED: exposed `clear_computation_cache()` which helps release the resources associated with a GPU context (the NuFHE `Context` objects call it automatically on destruction).
- ADDED: a `find_devices()` function to help with using multiple computation devices, and a corresponding keyword `device_id` for `Context` class constructor that uses its return values.
- ADDED: an example of multi-threaded multi-GPU usage.
- FIXED: a bug in `tlwe_noiseless_trivial()` occasionally leading to memory corruption.
- FIXED: a bug where `PerformanceParameters` and `PerformanceParametersForDevice` objects did not have a correct equality implementation, leading to unnecessary re-compilation of kernels.
- FIXED: compilation failing when `transforms_per_block` in `PerformanceParameters` is set too high.

1.3.2 0.0.1 (12 Oct 2018)

Initial version.

CHAPTER 2

Introduction

`nufhe` implements the fully homomorphic encryption algorithm from TFHE using CUDA and OpenCL. For the theoretical background one may refer to the works TFHE is based on:

- C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based.", Crypto 75-92 (2013);
- L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second.", Eurocrypt 617-640 (2015);
- I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds", Asiacrypt 3–33 (2016).

For more details check out this collection of papers on lattice cryptography.

Some additional performance improvements employed by `nufhe` are described in Implementation details.

CHAPTER 3

Installation

`nufhe` supports two GPU backends, CUDA (via PyCUDA) and OpenCL (via PyOpenCL). Neither of the backend packages can be installed by default, because, depending on the videocard, one of the platforms may be unavailable. Therefore, the user must pick one or more backends they want to use and request them explicitly during installation. A simple rule of thumb is to pick CUDA if you have an nVidia videocard, and OpenCL otherwise (although OpenCL will work with nVidia cards as well). Then `nufhe` can be installed using PyPi specifying the required extras.

For the CUDA backend:

```
$ pip install nufhe[pycuda]
```

For the OpenCL backend:

```
$ pip install nufhe[pyopencl]
```

For both CUDA and OpenCL backends:

```
$ pip install nufhe[pycuda,pyopencl]
```

CHAPTER 4

A short example

```python
import random
import nufhe

ctx = nufhe.Context()
secret_key, cloud_key = ctx.make_key_pair()

size = 32
bits1 = [random.choice([False, True]) for i in range(size)]
bits2 = [random.choice([False, True]) for i in range(size)]

ciphertext1 = ctx.encrypt(secret_key, bits1)
ciphertext2 = ctx.encrypt(secret_key, bits2)

reference = [not (b1 and b2) for b1, b2 in zip(bits1, bits2)]

vm = ctx.make_virtual_machine(cloud_key)
result = vm.gate_nand(ciphertext1, ciphertext2)
result_bits = ctx.decrypt(secret_key, result)

assert all(result_bits == reference)
```

4.1 Context

```python
ctx = nufhe.Context()
```

A context object represents an execution environment on a GPU (akin to a process), and is tied to a specific GPU device (if there are several available). The target device can be either selected interactively, or picked automaticall based on various filters; see the `Context` constructor for details.

Similar to a process, each context has its own memory space, and objects (keys and ciphertexts) from one context cannot be used in another one directly. One can transfer them between contexts via serialization/deserialization, see *NuFHESecretKey.dump()*, *NuFHECloudKey.dump()* and *LweSampleArray.dump()* for details.

4.2 Key pair

The next step is the creation of a secret and a cloud key. The former is used to encrypt plaintexts or decrypt cyphertexts; the latter is required to apply gates to encrypted data. Note that the cloud key can be rather large (of the order of 100Mb).

```
secret_key, cloud_key = ctx.make_key_pair()
```

make_key_pair() takes some keyword parameters that affect the security of the algorithm; the default values correspond to about 110 bits of security.

4.3 Encryption

Using the secret key we can encrypt some data with *encrypt()*. `nufhe` gates operate on bit arrays (either lists or `numpy` arrays):

```
size = 32
bits1 = [random.choice([False, True]) for i in range(size)]
bits2 = [random.choice([False, True]) for i in range(size)]

ciphertext1 = ctx.encrypt(secret_key, bits1)
ciphertext2 = ctx.encrypt(secret_key, bits2)
```

In this example we will test the NAND gate, so the reference result would be

```
reference = [not (b1 and b2) for b1, b2 in zip(bits1, bits2)]
```

4.4 Processing

Calculations are performed on a fully encrypted virtual machine created out of a cloud key:

```
vm = ctx.make_virtual_machine(cloud_key)
result = vm.gate_nand(ciphertext1, ciphertext2)
```

The output of a gate can be pre-initialized with *empty_ciphertext()* and passed to any gate function as a `dest` keyword parameter.

4.5 Decryption

After the processing, the person in possession of the secret key can decrypt the result with *decrypt()* and verify that the gate was applied correctly:

```
result_bits = ctx.decrypt(secret_key, result)
assert all(result_bits == reference)
```

4.6 GPU threads for the low-level API

`nufhe` uses Reikna as a backend for GPU operations, and all the low-level `nufhe` calls require a `reikna` Thread object, encapsulating a GPU context and a serialization queue for GPU kernel calls. It can be created interactively:

```
from reikna.cluda import cuda_api

thr = cuda_api().Thread.create(interactive=True)
```

where the user will be offered to choose between available platforms and videocards. Alternatively, one can pick an arbitrary available platform/device:

```
thr = cuda_api().Thread.create()
```

It is also possible to create a `Thread` using a known device, or an existing PyCUDA or PyOpenCL context. This is advanced usage, for those who plan to integrate `nufhe` into a larger GPU-based program. See the documentation for Thread and Thread.create() for details.

If one wants to use OpenCL instead of CUDA, `cuda_api` should be replaced with `ocl_api`. Alternatively, one can use `any_api` to select an arbitrary available backend.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

C

D

E

F

G

L

M

N

P

S

V