Faster Homomorphic Linear Transformations in HElib[⋆]

Shai Halevi[1] and Victor Shoup[1,2]

[1] IBM Research
[2] New York University

Abstract. HElib is a software library that implements homomorphic encryption (HE), with a focus on effective use of "packed" ciphertexts. An important operation (which is used in bootstrapping, as well as in other applications) is applying a known linear map to a vector of encrypted data. In this paper, we describe several algorithmic improvements that significantly speed up this operation: in our experiments, our new algorithms were 30–75 times faster than those currently implemented in HElib for typical parameters.
Our techniques also reduce the size of the large public evaluation key, often using 33%-50% less space than the previous HElib implementation. We also implemented a new tradeoff that enables a drastic reduction in size, maybe a 25x factor or more for some parameters, paying only a 2-4x factor in runtime (and giving up some parallelization opportunities).

Keywords. Homomorphic encryption, Implementation, Linear transformations

[⋆] Supported by the Defense Advanced Research Projects Agency (DARPA) and Army Research Office(ARO) under Contract No. W911NF-15-C-0236.

Table of Contents

1 Introduction

Homomorphic encryption (HE) [9, 3] enables performing arithmetic operations on encrypted data even without knowing the secret key. All contemporary HE schemes roughly follow the outline of Gentry's first candidate from 2009, where fresh ciphertexts are "noisy" to ensure security. This noise grows with every operation, until it becomes so large so as to cause decryption errors. This results in a "somewhat homomorphic" encryption scheme (SWHE) that can only evaluate low-depth circuits, such a scheme can be converted to a "fully homomorphic" encryption scheme (FHE) using bootstrapping. The most asymptotically efficient SWHE schemes are based on the hardness of ring-LWE. These scheme all use $R_p = \mathbb{Z}[X]/(F(X), p)$ as their native plaintext space, with F a cyclotomic polynomial and p an integer (usually a prime or prime power).

Smart and Vercauteren observed [11] that (for a prime p) an element in this native plaintext space can be used to encode (via Chinese Remaindering) a vector of values from a finite field $\mathbb{F}_{p^d}$, for some integer d that depends on F and p, and that operations on elements in R_p induce the corresponding entry-wise operation on the encoded vectors. This technique of encoding many plaintext elements from $\mathbb{F}_{p^d}$ in a single R_p element, which is then encrypted and manipulated homomorphically, is called "ciphertext packing", and the entries in the vector are called "plaintext slots." Gentry, Halevi, and Smart showed in [4] how to use special automorphisms on R_p (which were used for different purposes in [8] and [1]) to enable data movement between the slots.

HElib [7, 5, 6] is an open-source C++ library that implements the ring variant of the scheme due to Brakerski-Gentry-Vaikuntanathan [1], focusing on effective use of ciphertext packing. It includes an implementation of the BGV scheme itself with all its basic homomorphic operations, as well as higher-level procedures for data-movement, simple linear algebra, bootstrapping, etc. One can think of the lower levels of HElib as providing a "hardware platform", defining a set of operations that can be applied homomorphically. These operations include entry-wise additions and multiplications operations on the vector of plaintext values, as well as data movement, making this "platform" a SIMD environment.

Our Results. In this work, we improve performance of core linear algebra algorithms in HElib that apply publicly known linear transformations to encrypted vectors. We implemented these new algorithms in an experimental "fork" of HElib. For typical, realistic parameter settings, our new algorithms can run 30-75 times faster than those in the current implementation of HElib, where the exact speedup depends on myriad details.[3] Our implementation also exploits multiple cores, when available, to get even further speedups.

Our techniques also reduce the size of the large public evaluation key. In the old HElib implementation, the evaluation key typically consists of a large number of large "key switching matrices": Each of these "matrices" can take 1-4MB of space, and the implementation uses close to a hundred of them. Our new implementation reduces the number of key-switching matrices by 33–50% in some parameter settings (that arise fairly often in practice), while at the same time improves the running time. Moreover, a new tradeoff that we implemented enables a drastic reduction in the number of matrices (sometimes as few as four or six matrices overall), for a small prince of only 2-4x in performance. This space efficient variation, however, is inherently sequential, as opposed to our other procedure than can be easily parallelized.

[3] One could also consider algorithms that apply encrypted linear transformations to encrypted vectors; some of our new algorithmic techniques may apply to that problem as well; however, HElib does not implement such algorithms, and neither have we.

Applications. Linear transformations of encrypted vectors is a manifestly fundamental operation with many applications. For one example, HElib itself makes critical use of such transformations in its bootstrapping logic. As reported in [6], the bootstrapping routine can typically spend 25–40% of its time performing such transformations.

As another example, consider a *private information retrieval* protocol in which a client selects one value from a database of values held by a server, while hiding from the server which value was accessed. Using HE, one way to do this is for the server to encode each value as a column vector. The collection of all such values held by the server is thus encoded as a matrix M, where each column in M corresponds to one value. To access the ith value, the client can send to the server an encrypted unit vector v with 1 in the ith entry (or some other encrypted information from which the server can homomorphically compute such an encrypted unit vector). The server then homomorphically computes $M \times v$, which is an encryption of the selected column of M. The server sends the result to the client, who can decrypt it and recover the selected value.

Techniques. In the linear transformation algorithms currently implemented in HElib, the bulk of the time is spent moving data among the slots in the encrypted vector. As mentioned above, this is accomplished by using special automorphisms. The main cost of applying such an automorphism to a ciphertext is actually that of "key switching": after applying the automorphism to each ring element in the ciphertext (which is actually a very cheap operation), we end up with an encryption relative to the "wrong" secret key; by using data in the public key specific to this particular automorphism — a so-called "key switching matrix" — we can convert the ciphertext back to one that is an encryption relative to the "right" secret key. So the main goals in improving performance are to reduce the number of automorphisms, and to reduce the cost of each automorphism.

- To reduce the number of automorphisms, we introduce a "baby-step/giant-step" strategy for computing all of the required automorphisms. This strategy generalizes a similar idea that was used in [6] in the context of bootstrapping. This strategy by itself speeds up the computation by a factor of 15–20 in typical settings. See Section 4.1.
- We further reduce the number of automorphisms by refactoring a number of computations, more aggressively exploiting the algebraic properties of the automorphisms that we use. See Section 4.4.
- To reduce the cost of each automorphism, we introduce a new technique for "hoisting" the expensive parts of these operations out of the main loop.[4] Our main observation is that applying many automorphisms to the same ciphertext v can be done faster than applying each one separately. Instead we can perform an expensive pre-computation that depends only on v (but not the automorphisms themselves), and this pre-computation makes each automorphism much cheaper (typically, 6–8 times faster). See sections 4.2 and 5.
- Recall that key switching matrices are a part of the public key, we note that they consume quite a lot of space (typically several megabytes per matrix), so keeping their numbers down is desirable. In the current implementation of HElib, there can easily be several hundred such matrices in the public key. We introduce a new technique that reduces the number of key-switching matrices by 33–50% in some parameter settings (that arise fairly often in practice), while at the same time improves the running time of our algorithms. See Section 4.3.

[4] "Hoisting" is a term used in compiler optimization to describe the action of "hoisting" a computation out of a loop, so that it is only performed once, instead of in every loop iteration.

- We introduce yet another technique that drastically reduces the number of key-switching matrices to a very small number (less than 10), but comes at a cost in running time (typically 2–4 times more slowly as our fastest algorithms), and cannot be parallelized.[5] Achieving this reduction in key-switching storage without too much degradation in running time requires some new algorithmic ideas. See Section 4.5.

Outline. The rest of the paper is organized as follows.

- In Section 2, we introduce notation and terminology, and review the basics of the BGV cryptosystem, including ciphertext packing and automorphisms.
- In Section 3, we review the basic ideas underlying the current algorithms in HElib for applying linear transformations homomorphically. We focus on restricted linear transformations, the "one-dimensional" transformations MatMul1D and BlockMatMul1D. It turns out that considering these restricted transformations is sufficient: they can be used directly in applications such as bootstrapping, and can be easily be used to implement more general linear transformations.
- In Section 4, we give a more detailed overview of our new techniques.
- In Section 5, we give more of the details of our new hoisting technique.
- In Section 6, we present all of our new algorithms for MatMul1D and BlockMatMul1D in detail.
- In Section 7, we describe how to use algorithms for MatMul1D and BlockMatMul1D for more general linear transformations.
- In Section 8, we report on the performance of the implementation of our new algorithms.

2 Notations and Background

For a positive modulus $q \in \mathbb{Z}_{>0}$, we identify the ring $\mathbb{Z}_q$ with its representation as integers in $[-q/2, q/2)$ (except for $q = 2$ where we use $\{0, 1\}$). For integer z, we denote by $[z]_q$ the reduction of z modulo q into the same interval. This notation extends to vectors and matrices coordinate-wise, and to elements of other algebraic groups/rings/fields by considering their coefficients in some convenient basis (e.g., the coefficient of polynomials in the power basis when talking about $\mathbb{Z}[X]$). The norm of a ring element $\|a\|$ is defined as the norm of its coefficient vector in that basis.[6]

2.1 The BGV Cryptosystem

The BGV ring-LWE-based scheme [2] is defined over a ring $R \stackrel{\text{def}}{=} \mathbb{Z}[X]/(\Phi_m(X))$, where $\Phi_m(X)$ is the mth cyclotomic polynomial. For an arbitrary integer modulus N (not necessarily prime) we denote the ring $R_N \stackrel{\text{def}}{=} R/NR$.

As implemented in HElib, the native plaintext space of the BGV cryptosystem is R_{p^r} for a prime power p^r. The scheme is parametrized by a sequence of decreasing moduli $q_L \gg q_{L-1} \gg \cdots \gg q_0$, and an "$i$th level ciphertext" in the scheme is a vector $v \in R_{q_i}^2$. Secret keys are elements $s \in R$ with "small" coefficients (chosen in $\{0, \pm 1\}$ in HElib), and we view s as the second element of the 2-vector $\mathsf{sk} = (1, s) \in R^2$. A level-$i$ ciphertext $v = (p_0, p_1)$ encrypts a plaintext element $\alpha \in R_{p^r}$ with respect to $\mathsf{sk} = (1, s)$ if $[\langle \mathsf{sk}, v\rangle]_{q_i} = [p_0 + s \cdot p_1]_{q_i} = \alpha + p^r \cdot \epsilon$ (in R) for some "small" error term, $\|\epsilon\| \ll q_i/p^r$.

[5] While the "high level" operations using this technique are inherently sequential, the lower-level routines in HElib will still exploit multiple cores, if available. Such low-level parallelism are usually less effective, however.

[6] The difference between the norm in the different bases is not very important for the current work.

The error term grows with homomorphic operations of the cryptosystem, and switching from q_{i+1} to q_i is used to decrease the error term roughly by the ratio q_{i+1}/q_i. Once we have a level-0 ciphertext v, we can no longer use that technique to reduce the noise. To enable further computation, we need to use Gentry's bootstrapping technique [3]. In HElib, each q_i is a product of small (machine-word sized) primes.

2.2 Encoding Vectors in Plaintext Slots

As observed by Smart and Vercauteren [11], an element of the native plaintext space $\alpha \in R_{p^r}$ can be viewed as encoding a vector of "plaintext slots" containing elements from some smaller ring extension of $\mathbb{Z}_{p^r}$ via Chinese remaindering. In this way, a single arithmetic operation on α corresponds to the same operation applied component-wise to all the slots.

Specifically, suppose the factorization of $\Phi_m(X)$ modulo p^r is $\Phi_m(X) \equiv F_1(X) \cdots F_\ell(X) \pmod{p^r}$, where each F_i has the same degree d, which is equal to the order of p modulo m, so that $\ell = \phi(m)/d$. (This factorization can be obtained by factoring $\Phi_m(X)$ modulo p, followed by Hensel lifting.) Then we have the isomorphism $R_{p^r} \cong \bigoplus_{i=1}^{\ell}(\mathbb{Z}[X]/(p^r, F_i(X)))$.

Let us now denote $E \overset{\text{def}}{=} \mathbb{Z}[X]/(p^r, F_1(X))$, and let ζ be the residue class of X in E, which is a principal mth root of unity, so that $E = \mathbb{Z}/(p^r)[\zeta]$. The rings $\mathbb{Z}[X]/(p^r, F_i(X))$ for $i = 1, \ldots, \ell$ are all isomorphic to E, and their direct product is isomorphic to R_{p^r}, so we get an isomorphism between R_{p^r} and E^ℓ. HElib makes extensive use of this isomorphism, using it to encode an ℓ-vector of elements in E as an element of the native plaintext space R_{p^r}. Addition and multiplication of ciphertexts act on all ℓ slots of the corresponding plaintext in parallel.

2.3 Hypercube structure and one-dimensional rotations

Beyond addition and multiplications, we can also manipulate elements in R_{p^r} using a set of automorphisms on R_{p^r} of the form

$$\theta_t : R_{p^r} \longrightarrow R_{p^r}, \quad a(X) \longmapsto a(X^t) \pmod{(p^r, \Phi_m(X))}.$$

for $t \in \mathbb{Z}_m^*$. Since each θ_t is an automorphism, it distributes over addition and multiplication, i.e., $\theta_t(\alpha + \beta) = \theta_t(\alpha) + \theta_t(\beta)$ and $\theta_t(\alpha\beta) = \theta_t(\alpha)\theta_t(\beta)$. Also, these automorphisms commute with one another, i.e., $\theta_t\theta_{t'} = \theta_{tt'} = \theta_{t'}\theta_t$. Moreover, for any integer i, we have $\theta_t^i = \theta_{t^i}$.

We can homomorphically apply such an automorphism by applying it to the individual ciphertext components and then performing "key switching" (see [2, 4]). In somewhat more detail, a ciphertext in HElib consists of two "parts," each an element of R_q for some q. Applying the same automorphism (defined in R_q) to the two parts, we get a ciphertext with respect to a different secret key. In order to do anything more with this ciphertext, we usually have to convert it back to a ciphertext with respect to the original secret key. In order to do this, the public-key must contain data specific to the automorphism θ_t, called a "key switching matrix".[7] We will discuss this key-switching operation in more detail below in Section 5.

As discussed in [4], these automorphisms induce a hypercube structure on the plaintext slots, that depends on the structure of the group $\mathbb{Z}_m^*/\langle p\rangle$. Specifically, HElib keeps a hypercube basis

[7] Note that this "key switching" technique is a generalization of that used to allow multiplication of ciphertexts.

$g_1, \ldots, g_n \in \mathbb{Z}_m^*$ with orders $D_1, \ldots, D_n \in \mathbb{Z}_{>0}$, and then defines the set of representatives for $\mathbb{Z}_m^*/\langle p \rangle$ as

$$\{g_1^{e_1} \cdots g_n^{e_n} : 0 \leq e_s < D_s,\ s = 1, \ldots, n\}.$$

Thus, the slots are in one-to-one correspondence with tuples $(e_1, \ldots, e_n)$ with $0 \leq e_s < D_s$. This induces an n-dimensional hypercube structure on the plaintext space. If we fix $e_1, \ldots, e_{s-1}, e_{s+1}, \ldots, e_n$, and let e_s range over $0, \ldots, D_s - 1$, we get a set of D_s slots, which we refer to as a *hypercolumn in dimension* s (and there are ℓ/D_s such hypercolumns).

Using automorphisms, we can efficiently perform rotations in any dimension; a *rotation by* i *in dimension* s maps a slot corresponding to $(e_1, \ldots, e_s, \ldots, e_n)$ to the slot corresponding to $(e_1, \ldots, e_s + i \bmod D_s, \ldots, e_n)$. In other words, it rotates each hypercolumn in dimension s by i. We denote by ρ_s the rotation-by-1 operation in dimension s. Observe that ρ_s^i is the rotation-by-i operation in dimension s.

We can implement ρ_s^i by applying either one or two of the automorphisms $\{\theta_t\}_{t \in \mathbb{Z}_m^*}$ defined above. If the order of g_s in $\mathbb{Z}_m^*$ is D_s, then we get by with just a single automorphism, since

$$\rho_s^i(\alpha) = \theta_{g_s^i}(\alpha). \tag{1}$$

In this case, we call s a "good dimension".

If the order of g_s in $\mathbb{Z}_m^*$ is different from D_s, then we call s a "bad dimension", and we need to implement this rotation using two automorphisms. Specifically, we use a constant "0-1 mask value" μ that selects some slots and zeros-out the others, and use the two automorphisms $\psi \overset{\text{def}}{=} \theta_{g_s^i}$ and $\psi^* \overset{\text{def}}{=} \theta_{g_s^{i-D}}$. Then we have

$$\rho_s^i(\alpha) = \psi(\mu \cdot \alpha) + \psi^*((1 - \mu) \cdot \alpha). \tag{2}$$

The idea is roughly as follows. Even though ψ does not act as a rotation by i in dimension s, it does act as the desired rotation if we restrict it to inputs with zeros in each slot whose coordinate in dimension s is at least $D - i$. Similarly, ψ^* acts as the desired rotation if we restrict it to inputs with zeros in each slot whose coordinate in dimension s is less than $D - i$. This tells us that μ should have a 1 in all slots whose coordinate in dimension s is less than $D - i$, and a 0 in all other slots. Note also that

$$\rho_s^i(\alpha) = \mu' \cdot \psi(\alpha) + (1 - \mu') \cdot \psi^*(\alpha), \tag{3}$$

where $\mu' = \psi(\mu)$ is a mask with a 1 is all slots whose coordinate in dimension s is at least i, and a 0 in all other slots. This formulation will be convenient in some of the algorithms we present.

2.4 Frobenius and linearized polynomials

We define the automorphism $\sigma \overset{\text{def}}{=} \theta_p$, which is the Frobenius map on R_{p^r} (where θ_p is one of the automorphisms defined in Section 2.3). It acts on each slot independently as the Frobenius map σ_E on E, which sends ζ to ζ^p and leaves elements of $\mathbb{Z}_{p^r}$ fixed. (When $r = 1$, σ is the same as the pth power map on E.) For any $\mathbb{Z}_{p^r}$-linear transformation on E, denoted M, there exist unique constants $\lambda_0, \ldots, \lambda_{d-1} \in E$ such that $M(\eta) = \sum_{j=0}^{d-1} \lambda_j \sigma_E^j(\eta)$ for all $\eta \in E$. When $r = 1$, this follows from the general theory of linearized polynomials (see, e.g., Theorem 10.4.4 on p. 237 of [10]), but the same results are easily seen to hold for $r > 1$ as well. These constants are readily computable by solving a system of equations mod p^r.

Using linearized polynomials, we may effectively apply a fixed linear map to each slot of a plaintext element $\alpha \in R_{p^r}$ (either the same or different maps in each slot) by computing $\sum_{j=0}^{d-1} \kappa_j \sigma^j(\alpha)$, where the κ_j's are R_{p^r}-constants obtained by embedding appropriate E-constants in the slots.

2.5 Key switching strategies

The total number of automorphisms is $\phi(m)$, which is typically many thousands, so it is not very practical to store all possible key switching matrices in the public key: each such matrix typically occupies a few megabytes of storage, and storing all of them will consume hundreds of gigabytes. Therefore, we consider strategies that trade off space for time with respect to key switching matrices.

For almost all applications, we only need the key switching matrices for one-dimensional rotations in each dimension, as well as for the Frobenius map (and its powers). For a fixed dimension $s = 1, \ldots, n$ of size $D \overset{\text{def}}{=} D_s$ with generator $g \overset{\text{def}}{=} g_s$, consider the automorphism $\theta \overset{\text{def}}{=} \theta_{g_s}$. In the original implementation of HElib, one of two key switching strategies for dimension s are used.

Full: We store key switching matrices for θ^i for $i = 0, \ldots, D-1$. If s is a "bad dimension", we additionally store key switching matrices for θ^{-i} for $i = 1, \ldots, D-1$.

Baby-step/giant-step: We store key switching matrices for θ^j with $j = 1, \ldots, g-1$, where $g \overset{\text{def}}{=} \lceil \sqrt{D} \rceil$ (the "baby steps"), as well as for θ^{gk} with $k = 1, \ldots, h-1$, where $h \overset{\text{def}}{=} \lceil D/g \rceil$ (the "giant steps"). If s is a "bad dimension", we additionally store key switching matrices for θ^{-gk} with $k = 1, \ldots, h$ (negative "giant steps").

Using the full strategy, any rotation in dimension s can be implemented using a single automorphism and key switching if s is a good dimension, and using two automorphisms and key switchings if s is a bad dimension.

Using the baby-step/giant-step strategy, any rotation in dimension s can be implemented using at most two automorphisms and key switchings if s is a good dimension, and using at most four automorphisms and key switchings if s is a bad dimension. The idea is that to compute $\theta^i(v)$, for a given $i = 0, \ldots, D-1$, we can write $i = j + gk$, so that to compute $\theta^i(v)$, we first compute $w = \theta^{gk}(v)$, which takes one automorphism and a key switching, and then compute $\theta^j(w)$, which takes another automorphism and key switching.

These two strategies give us a time/space trade-off: although it slows down the computation time by a factor of two, the baby-step/giant-step strategy requires space for just $O(\sqrt{D})$ key switching matrices, rather than the $O(D)$ key switching matrices required by the full strategy.

The same two strategies can be used to store key switching matrices for powers of the Frobenius map, so that any power of the Frobenius map can be computed using either one or two automorphisms. Indeed, it is convenient to think of the powers of the Frobenius map as defining an additional (effectively "good") dimension.

The default behavior of HElib is to use the full key-switching strategy for "small" dimensions (of size at most 50), and the baby-step/giant-step strategy for larger dimensions.

3 Matrix multiplication — basic ideas

In [5], it is observed that we can multiply a matrix $M \in E^{\ell \times \ell}$ by a column vector $v \in E^{\ell \times 1}$ by computing

$$Mv = M_0 v_0 + \cdots + M_{\ell-1} v_{\ell-1}, \tag{4}$$

where each v_i is the vector obtained by rotating the entries of v by i positions, and each M_i is a diagonal matrix containing one diagonal of M.

3.1 MatMul1D: one-dimensional E-linear transformations

In many applications, such as the recryption procedure in [6], instead of a general E-linear transformation on R_{p^r}, we only need to work with a *one-dimensional* E-linear transformation that acts independently on the individual hypercolumns of a single dimension $s = 1, \ldots, n$. We can adapt the diagonal decomposition of Eqn. (4) to this setting using appropriate rotation maps on the slots of R_{p^r}. Let $\rho \overset{\text{def}}{=} \rho_s$ be the rotation-by-1 map in dimension s, and let $D \overset{\text{def}}{=} D_s$ be the size of dimension s. If T is one-dimensional E-linear transformation on R_{p^r}, then for every $v \in R_{p^r}$, we have

$$T(v) = \sum_{i=0}^{D-1} \kappa_i \cdot \rho^i(v), \tag{5}$$

where the κ_i's are constants in R_{p^r} determined by T, obtained by embedding appropriate constants in E in each slot. Eqn. (5) translates directly into a simple homomorphic evaluation algorithm, just by applying the same operations to a ciphertext encrypting v. In a straightforward implementation, in a good dimension, the computational cost is about D automorphisms and D constant-ciphertext multiplications, and the noise cost is a single constant-ciphertext multiplication. In bad dimensions, all of these costs would essentially double. In practice, if the constants have been pre-computed, the computation cost of the constant-ciphertext multiplications is negligible compared to that of the automorphisms.

One of our main goals in this paper is to dramatically improve upon the computational cost for performing such a MatMul1D operation.

3.2 BlockMatMul1D: one-dimensional $\mathbb{Z}_{p^r}$-linear transformations

In some applications (again, including the recryption procedure in [6]), instead of applying an E-linear transformation, we need to apply a $\mathbb{Z}_{p^r}$-linear map. Again, we focus on *one-dimensional* $\mathbb{Z}_{p^r}$-linear maps that act independently on the hypercolumns of a single dimension.

We can still use the same diagonal decomposition as in Eqn. (4), except that the entries in the diagonal matrices are no longer elements of E, but rather, $\mathbb{Z}_{p^r}$-linear maps on E. These maps may be encoded using linearized polynomials, as in Section 2.4. Therefore, if T is one-dimensional $\mathbb{Z}_{p^r}$-linear transformation on R_{p^r}, then for every $v \in R_{p^r}$, we have

$$T(v) = \sum_{i=0}^{D-1} \sum_{j=0}^{d-1} \kappa_{i,j} \cdot \sigma^j\big(\rho^i(v)\big), \tag{6}$$

where the $\kappa_{i,j}$'s are constants in R_{p^r} determined by T.

A naive homomorphic implementation of the formula from Eqn. (6) takes $O(dD)$ automorphisms, but as shown in [6], this can be reduced to $O(d + D)$ automorphisms. In this paper, we will also present significant improvements to the BlockMatMul1D algorithm in [6], although they are not as dramatic as our improvements to the MatMul1D algorithm.

4 Overview of algorithmic improvements

4.1 Baby-step/giant-step multiplication

As already mentioned, [6] introduces a technique that reduces the number of automorphisms needed to implement $\mathsf{BlockMatMul1D}$ in dimension s from $O(dD)$ to $O(d+D)$, where $D \overset{\text{def}}{=} D_s$ is the size of the dimension, and d is the order of p mod m. A very similar idea, essentially a baby-step/giant-step technique, can be used to reduce the number of automorphisms needed to implement $\mathsf{MatMul1D}$ in dimension s from $O(D)$ to $O(\sqrt{D})$. See Section 6 for details.

This technique is distinct from the baby-step/giant-step key switching strategy discussed above in Section 2.5. However, for best results, the two techniques should be combined in a way that harmonizes the baby-step/giant-step thresholds.

4.2 Hoisting

As we have seen, in many situations, we want to compute $\psi(v)$ for a fixed ciphertext v and many automorphisms ψ. Assuming we have key switching matrices for each automorphism ψ, the dominant cost of computing all of these values is that of performing one key-switching operation for each ψ. Our "hoisting" technique is a method that refactors the computation, performing a pre-computation that only depends on v, and whose computational cost is roughly equivalent to a single key-switching operation. After performing this pre-computation, computing $\psi(v)$ for any individual ψ is much faster than a single key-switching operation (typically, around 6–8 times faster). We describe this idea in more detail below in Section 5.

4.3 Better key switching strategies in bad dimensions

Recall from Section 2.5 that with the "full" key-switching strategy, in a bad dimension, we stored key-switching matrices for the automorphisms θ^i, with $i = -(D-1), \ldots, -1, 1, \ldots, D-1$. To perform a rotation by i on v in the given dimension, we need to compute $\theta^i(v)$ and $\theta^{i-D}(v)$, and so with these key-switching matrices available, we need to perform two automorphisms and key switchings. However, we do not really need all of these negative-power key switching matrices. In fact, we can get by with key-switching matrices just for θ^i, with $i = 1, \ldots, D-1$, and for θ^{-D}. To perform a rotation by i on v in the given dimension, we can compute $w = \theta^i(v)$ and $\theta^{-D}(w) = \theta^{i-D}(v)$. So again, we need to perform two automorphisms and key switchings. This cuts the number of key-switching matrices in half without a significant increase in running time. Moreover, this key-switching strategy aligns well with the strategy discussed below for decoupling rotations and automorphisms in bad dimensions.

Similarly, for the baby-step/giant-step key-switching strategy in a bad dimension, we just store a key-switching matrix for θ^{-D}, rather than for all the negative "giant steps". This cuts down the number of key-switching matrices by a third. Moreover, the number of key switchings we need to perform per rotation is only 3 (instead of 4).

4.4 Decoupling rotations and automorphisms in bad dimensions

Recall that by Eqn. (3), a rotation by i on a ciphertext v in a given bad dimension can be implemented as $\mu\theta^i(v) + (1-\mu)\theta^{i-D}(v)$, where μ is a "mask" (a constant with a 0 or 1 encoded in each slot). It turns out that in our matrix-vector computations, it is best to work directly with this implementation, and algebraically refactor the computation to improve both running time and noise. This refactoring exploits the fact that θ is an automorphism. See Section 6 for details.

4.5 A Horner-like rule with application to a minimal key-switching strategy

We introduce a new key-switching strategy that reduces the storage requirements even further, to just 1, 2, or 3 key-switching matrices per dimension. This, combined with a simple algorithmic idea, allow us to implement a variant of the baby-step/giant-step multiplication strategy that does not run too much more slowly than when using the full or baby-step/giant-step key-switching strategy. To do this, we observe that if we need to compute $\sum_{i=0}^{h-1} \psi^i(v_i)$, where ψ is some automorphism and the v_i's are ciphertexts, we can do this using Horner's rule, provided we have a key-switching matrix just for ψ. Specifically, we can compute

$$\sum_{i=0}^{h-1} \psi^i(v_i) = \psi\big(\cdots \psi\big(\psi(v_{h-1}) + v_{h-2}\big) + \cdots \big) + v_0,$$

setting $w_{h-1} \leftarrow v_{h-1}$ and $w_{i-1} \leftarrow \psi(w_i) + v_{h-1}$ for $i = h-1, \ldots, 1$, and outputting w_0.

4.6 Exploiting multi-core platforms

With the exception of the minimal key-switching strategy discussed above, all our other algorithms are very amenable to parallelization. We thus implemented them so as to exploit multiple cores, when available.

5 Hoisting

A ciphertext in HElib is a vector $v = (p_0, p_1) \in R_q^2$, with each "part" p_0, p_1 represented in a DoubleCRT format (i.e., both integer and polynomial CRT) [7]. We recall the steps in the computation of each $\psi(v)$, as implemented in HElib.

1. **Automorphism step:** We first apply the automorphism to each part of v, computing $p'_j \leftarrow \psi(p_j)$ for $j = 0, 1$.

   Applying an automorphism to a DoubleCRT object is a fast, linear time operation, so this step is cheap. If $v = (p_0, p_1)$ decrypts to α under the secret key $\mathsf{sk} = (1, s)$, then $v' = (p'_0, p'_1)$ decrypts to $\psi(\alpha)$ under the secret key $\mathsf{sk}' = (1, \psi(s))$. We next have to perform a "relinearization" operation which converts v' back to a ciphertext that decrypts to $\psi(\alpha)$ under the original secret key sk. This operation can itself be broken down into two steps:

2. **Break into digits step:** decompose p'_1 into "small" pieces: $p'_1 = \sum_k q'_k D_k$.
   Here, the D_k's are integer constants, and the pieces q'_k are elements of R of small norm. This operation is rather expensive, as it requires conversions between DoubleCRT and coefficient representations of elements in R_q.

3. Key switching step: compute the ciphertext $(p_0' + p_0'', p_1'')$, where

$$p_j'' = \sum_k q_k' A_{jk}, \quad (j = 0, 1).$$

Here, the A_{jk}'s are the "key switching matrices", namely, pre-computed elements in R_Q (for some larger Q) which are stored in the public key. The A_{jk}'s are stored in DoubleCRT format, so if we have the q_k' in the same DoubleCRT format then this operation is also a fast, linear time operation.

The key observation to our new technique is that we can reverse the order of the first two steps above, without affecting the correctness of the procedure. Namely our new procedure is as follows:

1. Break into digits step: decompose the original p_1 *before applying the automorphism* into "small" pieces: $p_1 = \sum_k q_k D_k$.

2. Automorphism step: compute $p_0' \leftarrow \psi(p_0)$, and $q_k' \leftarrow \psi(q_k)$ for each q_k. Namely, p_0' is computed just as before, but we apply the automorphism to the pieces q_k from above rather than to p_1 itself.

3. Key switching step: compute the ciphertext $(p_0' + p_0'', p_1'')$, where

$$p_j'' = \sum_k q_k' A_{jk}, \quad (j = 0, 1).$$

This is exactly the same computation as before.

The reasons that this works, is that (i) ψ is an automorphism (so it distributes over addition and multiplication), and (ii) applying ψ does not significantly change the norm of an element (cf. [8]). In a little more detail, correctness of the key-switching step depends only on the following two conditions on the q_k''s:

(a) $\sum_k q_q' D_k = \psi(p_1)$, and

(b) the q_k''s have low norm.

Condition (a) is satisfied in our new procedure since ψ is an automorphism (which acts as the identity on integers), and so

$$\psi(p_1) = \psi\big(\sum_k q_k D_k\big) = \sum_k \psi(q_k) D_k = \sum_k q_k' D_k.$$

Condition (b) is satisfied since the pieces q_k have small norm, and applying ψ to a ring element does not increase its norm significantly.

The new procedure is therefore just as effective as the old one, but now the expensive break-into-digits step can be preformed only once, as a pre-computation that depends only on v, rather than having to perform it for every automorphism ψ. The flip side is that we need to apply ψ to each one of the parts q_k instead of only once to p_1. But as we mentioned, this is a cheap operation.

5.1 Interaction with key-switching strategy

If we want to compute $\psi(v)$ for various automorphisms ψ, and we have key-switching matrices for all of the ψ's. then we can apply the above hoisting strategy directly. In some situations, what we want to do is compute $\theta^i(v)$ for $i = 0, \ldots, D-1$, where $\theta = \theta_{g_s}$ for some dimension s with generator $g_s \in \mathbb{Z}_m^*$, and where $D = D_s$ is the size of the dimension. If we are employing the baby-step/giant-step strategy for storing key-switching matrices, then we do not have all of the requisite key-switching matrices, so we cannot use the hoisting strategy directly. Instead, what we can do is the following. Since we have key-switching matrices for all of the giant steps θ^{gj}, for $j = 1, \ldots, h-1$, we can use hoisting to compute $\theta^{gj}(v)$ for all of the giant steps, and for each of these values, we perform the pre-computation (i.e., the break-into-digits step). Then, since we have key-switching matrices for all of the baby steps θ^k, for $k = 1, \ldots, g-1$, we can compute any value $\theta^{gj+k}(v)$ as $\theta^k(\ \theta^{gj}(v)\ )$, using the precomputed data for $\theta^{gj}(v)$ and the key-switching matrix for θ^k.

6 Algorithms for one-dimensional linear transformations

In this section, we describe in detail our algorithms for applying one-dimensional linear transformations to a ciphertext v. We fix a dimension $s = 1, \ldots, n$. Recall from Section 2.3 that $\rho \overset{\text{def}}{=} \rho_s$ is the rotation-by-1 map in dimension s, and that $D \overset{\text{def}}{=} D_s$ is the size of dimension s.

6.1 Logic for basic MatMul1D

Recall from Section 3.1 that for that MatMul1D calculation, we need to compute

$$w = \sum_{i \in [D]} \kappa(i) \rho^i(v),$$

where the $\kappa(i)$'s are constants in R_{p^r} that depend on the matrix.

If s is a good dimension, then ρ is realized with a single automorphism, $\rho = \theta \overset{\text{def}}{=} \theta_{g_s}$ where $g_s \in \mathbb{Z}_m^*$ is the generator for dimension s. We can easily implement this in a number of ways. For example, we can use the hoisting technique from Section 5 to compute all of the values $\theta^i(v)$ for $i \in [D]$. Alternatively, if we are using a minimal key-switching strategy (see Section 4.5), then with just a key-switching matrix for θ, we can compute the values $\theta^i(v)$ iteratively, computing $\theta^{i+1}(v)$ from $\theta^i(v)$ as $\theta(\theta^i(v))$.

6.2 Revised logic for bad dimensions

From Eqn. (3), if s is a bad dimension, then we have

$$\rho^i(v) = \mu(i)\theta^i(v) + \mu'(i)\theta^{i-D}(v), \tag{7}$$

where $\mu(i)$ is a "0-1 mask" and $\mu'(i) = 1 - \mu(i)$. As discussed in Section 4.4, it is useful to algebraically decouple the rotations and automorphisms in a bad dimension, which we can do as follows:

$$\begin{aligned} w &= \sum_{i \in [D]} \kappa(i) \rho^i(v) \\ &= \sum_{i \in [D]} \kappa(i) \big\{ \mu(i) \theta^i(v) + \mu'(i) \theta^{i-D}(v) \big\} \\ &= \sum_{i \in [D]} \kappa'(i) \theta^i(v) \quad + \quad \theta^{-D} \Bigg[\sum_{i \in [D]} \kappa''(i) \theta^i(v) \Bigg], \end{aligned}$$

where

$$\begin{aligned} \kappa'(i) &= \mu(i)\kappa(i) \quad \text{and} \\ \kappa''(i) &= \theta^D\{\mu'(i)\kappa(i)\}. \end{aligned}$$

To implement this, we have to compute $\theta^i(v)$ for all $i \in [D]$. This can be done using the same strategies as were discussed above in a good dimension, using either hoisting or iteration. The only other automorphism we need to compute is one evaluation of θ^{-D}. Note that with our new key-switching strategy (see Section 4.3), we always have available a key-switching matrix for θ^{-D}.

If we ignore the cost of pre-computing all the constants in DoubleCRT format, we see that the computational cost is roughly the same in both good and bad dimensions. This is because the time needed to perform all the constant-ciphertext multiplications is very small in comparison to the time needed to perform all the automorphisms. The cost in noise is also about the same, essentially, one constant-ciphertext multiplication.

6.3 Baby-step/giant-step logic

We now present the logic for a new baby-step/giant-step multiplication algorithm. As discussed above in Section 4.1, this idea is very similar to the BlockMatMul1D implementation described in [6]. Set $g = \lceil \sqrt{D} \rceil$ and $h = \lceil D/g \rceil$. We have:

$$\begin{aligned} w &= \sum_{i \in [D]} \kappa(i) \rho^i(v) \\ &= \sum_{j \in [g]} \sum_{k \in [h]} \kappa(j + gk) \rho^{j+gk}(v) \\ &= \sum_{k \in [h]} \rho^{gk} \Bigg[\sum_{j \in [g]} \kappa'(j + gk) \rho^j(v) \Bigg], \end{aligned}$$

where $\kappa'(j + gk) = \rho^{-gk}(\kappa(j + gk))$.

Algorithm 1. In a good dimension, where $\rho = \theta$, we can implement the above logic using the following algorithm.

1. For each $j \in [g]$, compute $v_j = \theta^j(v)$.
2. For each $k \in [h]$, compute
$$w_k = \sum_{j \in [g]} \kappa'(j + gk) v_j.$$

3. Compute
$$w = \sum_{k \in [h]} \theta^{gk}(w_k).$$

Step 1 of the algorithm can be implemented by hoisting, or if we are using a minimal key-switching strategy, by iteration. Also, if we using a minimal key-switching strategy, Step 3 can be implemented using the Horner-rule idea discussed in Section 4.5 — for this, we just need a key-switching matrix for θ^g. Otherwise, if we have key switching matrices for all of the ρ^{gk}'s, it is somewhat faster to apply all of these automorphisms independently, which is also amenable to parallelization.

6.4 Revised baby-step/giant-step logic for bad dimensions

Set $g = \lceil \sqrt{D} \rceil$ and $h = \lceil D/g \rceil$. Again, using Eqn. (7), and the idea of to algebraically decoupling the rotations and automorphisms in a bad dimension, we have:

$$\begin{aligned} w &= \sum_{i \in [D]} \kappa(i) \rho^i(v) \\ &= \sum_{i \in [D]} \kappa(i) \big\{ \mu(i) \theta^i(v) + \mu'(i) \theta^{i-D}(v) \big\} \\ &= \sum_{j \in [g]} \sum_{k \in [h]} \kappa(j+gk) \big\{ \mu(j+gk) \theta^{j+gk}(v) + \mu'(j+gk) \theta^{j+gk-D}(v) \big\} \\ &= \sum_{k \in [h]} \theta^{gk} \Bigg[\sum_{j \in [g]} \big\{ \kappa'(j+gk) \theta^j(v) + \kappa''(j+gk) \theta^{j-D}(v) \big\} \Bigg], \end{aligned}$$

where

$$\begin{aligned} \kappa'(j+gk) &= \theta^{-gk} \big\{ \mu(j+gk) \kappa(j+gk) \big\} \quad \text{and} \\ \kappa''(j+gk) &= \theta^{-gk} \big\{ \mu'(j+gk) \kappa(j+gk) \big\}. \end{aligned}$$

Based on this, we derive the following:

Algorithm 2.

1. Compute $v' = \theta^{-D}(v)$.
2. For each $j \in [g]$, compute $v_j = \theta^j(v)$ and $v'_j = \theta^j(v')$
3. For each $k \in [h]$, compute
$$w_k = \sum_{j \in [g]} \big\{ \kappa'(j+gk) v_j + \kappa''(j+gk) v'_j \big\}.$$
4. Compute
$$w = \sum_{k \in [h]} \theta^{gk}(w_k).$$

Step 2 of the algorithm can be implemented by hoisting, or if we are using a minimal key-switching strategy, by iteration. Also, if we using a minimal key-switching strategy, Step 4 can be implemented using Horner's rule. As before, if we have key switching matrices for all of the ρ^{gk}'s, it is somewhat faster to apply all of these automorphisms independently, which is also amenable to parallelization.

Based on experimental data, we find that using the baby-step/giant-step multiplication algorithms are faster in dimensions for which we are using a baby-step/giant-step key-switching strategy. Moreover, even if we are using the full key-switching strategy, and we have all key-switching matrices for that dimensions available, the baby-step/giant-step multiplication algorithms are still faster in very large dimensions (say, on the order of several hundred).

6.5 Alternative revised baby-step/giant-step logic for bad dimensions

We considered, implemented, and tested an alternative algorithm, which was found to be slightly slower and was hence disabled. It proceeds as follows: Set $g = \lceil \sqrt{D} \rceil$ and $h = \lceil D/g \rceil$.

$$\begin{aligned}
w &= \sum_{i \in [D]} \kappa(i) \rho^i(v) \\
&= \sum_{i \in [D]} \kappa(i) \big\{ \mu(i) \theta^i(v) + \mu'(i) \theta^{i-D}(v) \big\} \\
&= \sum_{j \in [g]} \sum_{k \in [h]} \kappa(i) \big\{ \mu(j+gk) \theta^{j+gk}(v) + \mu'(j+gk) \theta^{j+gk-D}(v) \big\} \\
&= \sum_{k \in [h]} \theta^{gk} \Bigg[\sum_{j \in [g]} \kappa'(j+gk) \theta^j(v) \Bigg] \quad + \quad \theta^{-D} \left(\sum_{k \in [h]} \theta^{gk} \Bigg[\kappa''(j+gk) \theta^j(v) \big\} \Bigg] \right),
\end{aligned}$$

where

$$\begin{aligned}
\kappa'(j+gk) &= \theta^{-gk} \left\{ \mu(j+gk) \kappa(j+gk) \right\} \quad \text{and} \\
\kappa''(j+gk) &= \theta^{D-gk} \left\{ \mu'(j+gk) \kappa(j+gk) \right\}.
\end{aligned}$$

Based on this, we derive the following:

Algorithm 3.

1. For each $j \in [g]$, compute $v_j = \theta^j(v)$
2. For each $k \in [h]$, compute
$$u_k = \sum_{j \in [g]} \kappa'(j+gk) v_j \quad \text{and} \quad u'_k = \sum_{j \in [g]} \kappa''(j+gk) v'_j.$$
3. Compute
$$u = \sum_{k \in [h]} \theta^{gk}(u_k) \quad \text{and} \quad u' = \sum_{k \in [h]} \theta^{gk}(u'_k).$$
4. Compute
$$w = u + \theta^{-D}(u').$$

6.6 BlockMatMul1D logic

Recall from Section 3.2 that for the BlockMatMul1D calculation, we need to compute

$$\begin{aligned} w &= \sum_{j\in[d]} \sum_{i\in[D]} \kappa(i,j)\sigma^j(\rho^i(v)) \\ &= \sum_{j\in[d]} \sigma^j \Bigg[\sum_{i\in[D]} \kappa'(i,j)\rho^i(v) \Bigg], \end{aligned}$$

where $\kappa'(i,j) = \sigma^{-j}(\kappa(i,j))$. Here, σ is the Frobenius automorphism. This strategy is very similar to the baby-step/giant-step strategy used for the MatMul1D computation.

Algorithm 4. In a good dimension, where $\rho = \theta$, we can implement the above logic using the following algorithm.

1. Initialize an accumulator $w_j = 0$ for each $j \in [d]$.
2. For each $i \in [D]$:
   (a) compute $v_i = \theta^i(v)$;
   (b) for each $j \in [d]$, add $\kappa'(i,j)v_i$ to w_j.
3. Compute
$$w = \sum_{j\in[d]} \sigma^j(w_j).$$

Step 2(a) of the algorithm can be implemented by hoisting, or if we are using a minimal key-switching strategy, by iteration. Also, if we are using a minimal key-switching strategy, Step 3 can be implemented using Horner's rule, using just a key-switching matrix for σ. If we have key switching matrices for all of the σ^j's, it is somewhat faster to apply all of these automorphisms independently, which is also amenable to parallelization.

Typically, D is much larger than d. Assuming we are using the hoisting technique in Step 2(a), it is much faster to perform Step 2(a) on the dimension of larger size D, and to perform Step 3 on the dimension of smaller size d. Indeed, the amortized cost of computing each of the d automorphisms in Step 3 is much greater than the amortized cost of computing each of the D automorphisms (via hoisting) in Step 2(a). Note that in our actual implementation, if it turns out that D is in fact smaller than d, then we switch the roles of θ and σ.

Observe that we store d accumulators $u_0, \ldots, u_{d-1}$, rather than store the intermediate values $v_0, \ldots, v_{D-1}$. Either strategy would work, but assuming D is much larger than d, we save space with this strategy (even though it is slightly more challenging to parallelize).

6.7 Revised BlockMatMul1D logic for bad dimensions

Again, using Eqn. (7) and the idea of algebraically decoupling rotations and automorphism, we have:

$$\begin{aligned} w &= \sum_{j\in[d]} \sum_{i\in[D]} \kappa(i,j)\sigma^j(\rho^i(v)) \\ w &= \sum_{j\in[d]} \sum_{i\in[D]} \kappa(i,j)\sigma^j \left\{\mu(i)\theta^i(v) + \mu'(i)\theta^{i-D}(v)\right\} \\ &= \sum_{j\in[d]} \sigma^j \Bigg[\sum_{i\in[D]} \kappa'(i,j)\theta^i(v) \Bigg] \quad + \quad \theta^{-D} \Bigg(\sum_{j\in[d]} \sigma^j \Bigg[\sum_{i\in[D]} \kappa''(i,j)\theta^i(v) \Bigg] \Bigg), \end{aligned}$$

where

$$\kappa'(i,j) = \sigma^{-j}(\kappa(i,j))\mu(i) \quad \text{and}$$
$$\kappa''(i,j) = \theta^{D}\{\sigma^{-j}(\kappa(i,j))\mu'(i)\}.$$

Based on this, we derive the following:

Algorithm 5.

```
1. Initialize accumulators $u_j = 0$ and $u'_j = 0$ for each $j \in [d]$.
2. For each $i \in [D]$:
   (a) compute $v_i = \rho^i(v)$;
   (b) for each $j \in [d]$, add $\kappa'(i,j)v_i$ to $u_j$ and add $\kappa''(i,j)v_i$ to $u'_j$
3. Compute
       $u = \sum_{j\in[d]} \sigma^j(u_j) \quad \text{and} \quad u' = \sum_{j\in[d]} \sigma^j(u'_j).$
4. Compute
       $w = u + \theta^{-D}(u').$
```

As above, Step 2(a) of the algorithm can be implemented by hoisting, or if we are using a minimal key-switching strategy, by iteration. Also, if we using a minimal key-switching strategy, Step 3 can be implemented using Horner's rule, using just a key-switching matrix for σ. Again, if it turns out that D is in fact smaller than d, then we switch the roles of θ and σ.

7 Algorithms for arbitrary linear transformations

So far, we have described algorithms for applying one-dimensional linear transformations to an encrypted vector, that is, E- or $\mathbb{Z}_{p^r}$-linear transformations that act independently on the hypercolumns in a single dimension (i.e, the MatMul1D and BlockMatMul1D operations introduced in Section 3). Many of the techniques we have introduced can be adapted to arbitrary linear transformations. However, from a software design point of view, we adopted a strategy of designing a simple reduction from the general case to the one-dimensional case. For some parameter settings, this approach may not be optimal, but it will usually be much faster than the current implementations of these operations in HElib.

We first consider the MatMulFull operation, which applies a general E-linear transformation to an encrypted vector. Here, an encrypted vector is a ciphertext whose corresponding plaintext is a vector with $\ell = \phi(m)/d$ slots. One can easily extend the MatMulFull operation to E-linear transformations on larger encrypted vectors that comprise several ciphertexts, although we have not yet implemented such an extension.

Recall from Section 2.3 that $\ell = D_1 \cdots D_n$, where for $s = 1, \ldots, n$, the size of dimension s is D_s, and ρ_s is the rotation-by-1 map on dimension s. In [5], it was observed that we can apply the MatMulFull operation to a ciphertext v by using a generalization of the simple rotation strategy we presented above in Eqn. (4). More specifically, if T is an E-linear transformation on R_{p^r}, then for every $v \in R_{p^r}$, we have

$$T(v) = \sum_{i_1\in[D_1]} \cdots \sum_{i_n\in[D_n]} \kappa_{i_1,\ldots,i_n} \cdot (\rho_n^{i_n} \cdots \rho_1^{i_1})(v), \tag{8}$$

where the $\kappa_{i_1,\ldots,i_n}$'s are constants in R_{p^r} determined by the linear transformation. For each $(i_1, \ldots, i_{n-1})$, there is a one-dimensional E-linear transformation $T'_{i_1,\ldots,i_{n-1}}$ that acts on dimension n, such that for every $w \in R_{p^r}$, we have

$$T'_{i_1,\ldots,i_{n-1}}(w) = \sum_{i_n \in [D_n]} \kappa_{i_1,\ldots,i_n} \cdot (\rho_n^{i_n} \cdots \rho_1^{i_1})(w).$$

Therefore, we can refactor Eqn. (8) as follows:

$$T(v) = \sum_{i_1 \in [D_n]} \cdots \sum_{i_{n-1} \in [D_{n-1}]} T'_{i_1,\ldots,i_{n-1}} \big\{ (\rho_{n-1}^{i_{n-1}} \cdots \rho_1^{i_1})(v) \big\}. \tag{9}$$

To implement Eqn. (9), we compute all of the rotations $(\rho^{i_{n-1}} \cdots \rho^{i_1})(v)$ using a simple recursive algorithm. The main type of operation performed here is to compute all of the rotations $\rho_s^{i_s}(w)$ for a given w, a given dimension, and for all $i_s \in [D_s]$. In a good dimension, where $\rho_s = \theta_{g_s}$, we can use hoisting (see Section 5) to speed things up, provided the required key-switching matrices are available, or sequentially if not. For bad dimensions, we can use the decoupling idea discussed in Section 4.4. Specifically, using Eqn. (7), if $\theta \overset{\text{def}}{=} \theta_{g_s}$, then

$$\rho_s^{i_s}(w) = \mu_{i_s} \theta^{i_s}(w) + (1 - \mu_{i_s}) \theta^{i_s - D_s}$$

for an appropriate mask μ_{i_s}. Then we can compute $w' = \theta^{-D_s}$, which requires a single key-switching using our new key-switching strategy (see Section 4.3). After this, we need to compute $\theta^{i_s}(w)$ and $\theta^{i_s}(w')$ for all $i_s \in [D_s]$, which again, can be done by hoisting or iteration, as appropriate.

The other main type of operation needed to implement Eqn. (9) is the application of all of the one-dimensional transformations $T'_{i_1,\ldots,i_{n-1}}$ in dimension n, for which we can use our improved implementation of MatMul1D.

The speedup over the current implementation in HElib will be roughly equal to the speedup of our new implementation of MatMul1D in dimension n. So to get the best performance, our implementation orders the dimensions to that D_n is the largest dimension size. If dimension n is a bad dimension, we also save on noise as well (we save noise equal to that of one constant-ciphertext multiplication). In many applications, it is desirable to choose parameters so that there is one very large dimension, and zero, one, or two very small dimensions — indeed, by default, HElib will choose parameters in this way. In this typical setting, the speedup for MatMulFull will be very significant.

Finally, we mention that the above techniques carry over in an obvious way to general $\mathbb{Z}_{p^r}$-linear transformations on R_{p^r}. As above, we there is a simple reduction from the general BlockMatMulFull operation to the one-dimensional BlockMatMul1D operation. The current implementation of BlockMatMulFull is not particularly well optimized, and because of this, the speedup we get is roughly equal to n times the speedup of our implementation of BlockMatMul1D, where, again, n is the number of dimensions in the underlying hypercube.

8 Timings

We now present some timing data that demonstrates the effectiveness of our new techniques. All of our testing was done on a machine with an Intel Xeon CPU, E5-2698 v3 @2.30GHz (which is a Haswell processor), featuring 32 cores and 250GB of main memory. The compiler was GCC version

4.8.5, and we used NTL version 10.5.0 and GMP version 6.0. We implemented our new techniques in an experimental "fork" of HElib.

Table 1 shows the running time (in seconds) for the old default behavior ("old def") and the new default behavior ("new def") for MatMul1D computations (see Section 3.1). We do this for various values of m defining a cyclotomic polynomial of degree $\phi(m)$. The quantity d is the order of p mod m (which represents the "size" of each slot), while the quantity D is the size of the dimension. A value of D marked with "*" denotes a "bad" dimension. Table 1 does not show the time taken to build the constants associated with a matrix or to convert them to DoubleCRT representation. One sees that for the large dimension of size 682 (which is a typical size for many applications), we get a speedup of 30 if it is a good dimension, and a speedup of 75 if it is bad. Speedups for smaller dimensions are less dramatic, but still quite significant.

Table 2 shows more detailed information on various implementation strategies, as well as the cost of precomputing matrix constants. The "build" column shows the time to build the constants associated with the matrix in a polynomial representation. The "conv" column shows that time required to convert these constants to DoubleCRT representation. The following columns show that time required to perform the matrix-vector multiplication, based on a variety of key switching and algorithmic strategies. The columns are labeled as "[MBF]/[BF][HN]", where

MBF: **M** is for **M**in KS strategy, **B** is for **B**aby-step/giant-step key-switching strategy, **F** is for Full key-switching strategy,

BF: **B** is for **B**aby-step/giant-step multiplication strategy, **F** is for **F**ull multiplication strategy,

HN: **H** is for **H**oisting, **N** is for **N**o hoisting.

As one can see from the data, the cost of converting constant to DoubleCRT representation can easily exceed the cost of the remaining operations, so it is essential that these conversions are done as precomputations, if at all possible.

Consider the first line in Table 2. Column B/BH represents the default behavior: baby-step/giant-step key switching (since it is a large dimension of size 682), baby-step/step-step multiplication, and hoisting (only the baby steps are subject to hoisting). The next column (B/BN) is the same, except the baby steps are not hoisted, which is why it is slower. Column B/FH shows what happens if we do not use baby-step/giant-step multiplication, and rely exclusively on hoisting (as in Section 5.1). One can see that for such a large dimension, this is not an optimal strategy. Column M/B shows what happens when we use the minimal key switching strategy (with baby-step/step-step multiplication). Even though it needs only two key switching matrices (rather than about 50), it is less that twice as slow as the best strategy (although it does not parallelize very well). The algorithm represented by column B/FN corresponds directly to the algorithm originally implemented in HElib. The next line in the table represents a bad dimension. We note that for bad dimensions, the algorithm originally implemented in HElib is about twice as slow as the one represented by column B/FN (this is why the timing data in Table 1 for bad dimensions is not equal to the numbers in column B/FN of Table 2).

Table 3 shows corresponding timing data for BlockMatMul1D computations (see Section 3.2). For good dimensions, the current implementation in HElib roughly corresponds to the non-hoisting strategy in our new implementation. So one can see that with hoisting we get a speedup of up to 4 times over the current implementation for large dimensions (but only about 1.5 for small dimensions). For large, bad dimensions, in the current implementation in HElib, the running time will be close to twice that of the non-hoisting strategy in our new implementation; therefore, the speedup in such dimensions is close to a factor 8.

Table 4 shows the effectiveness of parallelization using multiple cores. We show times for both MatMul1D and BlockMatMul1D, using 1, 4, and 16 threads. These times are for the default strategies, and do not show the time required to build the matrix constants or convert them to DoubleCRT representation. While the speedups do not quite scale linearly with the number of cores, they are clearly significant, with 16 cores giving roughly an 8X speedup in large dimensions and 4X speedup in small ones.

We do not present detailed results for the running times of our new implementation of MatMulFull and BlockMatMulFull, discussed in Section 7. However, our experiments indicate that the speedups predicted in Section 7 closely align with practice: the speedup for MatMulFull is about the same as our speedup for MatMul1D in the largest dimension; the speedup for BlockMatMulFull is roughly our speedup for BlockMatMul1D in the largest dimension, times the number of dimensions in the hypercube.

m	$\phi(m)$	d	D	old def	new def	speedup
15709	15004	22	682	69.28	2.22	31.20
15709	15004	22	$682^\star$	138.20	3.14	75.86
18631	18000	25	120	20.27	1.38	14.69
18631	18000	25	$120^\star$	39.97	1.69	23.65
24295	18816	28	42	3.18	0.51	6.24
24295	18816	28	$42^\star$	6.20	0.55	11.27

Table 1. MatMul1D: summary of old vs new

m	$\phi(m)$	d	D	build	conv	M/B	M/F	B/BH	B/BN	B/FH	B/FN	F/FH	F/FN
15709	15004	22	682	0.47	5.54	3.80	44.81	2.22	3.19	6.46	69.28	5.30	28.30
15709	15004	22	$682^\star$	0.56	11.07	5.93	44.86	3.14	5.03	7.33	69.70	5.94	29.16
18631	18000	25	120	0.08	1.96	2.43	13.81	1.38	2.04	2.36	20.27	1.29	8.70
18631	18000	25	$120^\star$	0.10	3.91	3.68	13.95	1.69	2.89	2.45	20.27	1.29	8.78
24295	18816	28	42	0.03	0.70	1.39	5.09	0.82	1.17	1.11	6.87	0.51	3.18
24295	18816	28	$42^\star$	0.04	1.39	2.17	5.09	0.95	1.64	1.20	6.94	0.55	3.20

Table 2. MatMul1D

m	$\phi(m)$	d	D	build	conv	M/	B/H	B/N	F/H	F/N
15709	15004	22	682	15.47	122.62	54.73	21.03	84.42	18.15	42.67
15709	15004	22	$682^\star$	17.31	246.89	64.98	36.81	99.84	32.41	57.07
18631	18000	25	120	2.44	49.59	18.83	9.84	27.90	6.88	14.66
18631	18000	25	$120^\star$	2.96	98.79	23.83	17.62	35.80	12.73	20.58
24295	18816	28	42	0.95	19.73	9.25	7.84	13.64	5.01	7.70
24295	18816	28	$42^\star$	1.15	39.72	13.49	14.73	20.45	9.65	12.47

Table 3. BlockMatMul1D

				MatMul1D			BlockMatMul1D		
m	$\phi(m)$	d	D	$nt = 1$	$nt = 4$	$nt = 16$	$nt = 1$	$nt = 4$	$nt = 16$
15709	15004	22	682	2.18	0.67	0.29	20.21	7.60	2.47
15709	15004	22	$682^\star$	3.14	0.97	0.42	35.50	12.17	4.70
18631	18000	25	120	1.35	0.49	0.20	7.97	2.49	1.03
18631	18000	25	$120^\star$	1.65	0.58	0.29	13.89	4.30	1.67
24295	18816	28	42	0.47	0.23	0.15	4.98	1.37	0.61
24295	18816	28	$42^\star$	0.51	0.22	0.14	9.51	2.67	1.08

Table 4. MatMul1D/BlockMatMul1D (multi-core)

References

1. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science (ITCS'12)*, 2012. Available at `http://eprint.iacr.org/2011/277`.
2. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):13, 2014.
3. C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.
4. C. Gentry, S. Halevi, and N. Smart. Fully homomorphic encryption with polylog overhead. In *"Advances in Cryptology - EUROCRYPT 2012"*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. Full version at `http://eprint.iacr.org/2011/566`.
5. S. Halevi and V. Shoup. Algorithms in HElib. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014, Part I*, pages 554–571. Springer, 2014. Long version at `http://eprint.iacr.org/2014/106`.
6. S. Halevi and V. Shoup. Bootstrapping for HElib. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.
7. S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. `https://github.com/shaih/HElib/`, Accessed September 2014.
8. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.
9. R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.
10. S. Roman. *Field Theory.* Springer, 2nd edition, 2005.
11. N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, 71(1):57–81, 2014. Early verion at `http://eprint.iacr.org/2011/133`.