HLG: A framework for computing graphs in Residue Number System and its application in Fully Homomorphic Encryption

Shuang Wu[1] and Chunhuan Zhao[2], Ye Yuan[1], Shuzhou Sun[2], Jie Li[2], Yamin Liu[1]

[1] Huawei International, Singapore [wu.shuang,yuanye44,liuyamin3]@huawei.com
[2] Huawei Technologies, China [zhaochunhuan,sunshuzhou,lijie303]@huawei.com

Abstract. Implementation of Fully Homomorphic Encryption (FHE) is challenging. Especially when considering hardware acceleration, the major performance bottleneck is data transfer. Here we propose an algebraic framework called Heterogenous Lattice Graph (HLG) to build and process computing graphs in Residue Number System (RNS), which is the basis of high performance implementation of mainstream FHE algorithms.
There are three main design goals for HLG framework:

- Design a dedicated IR (HLG IR) for RNS system, where splitting and combination of data placeholders has practical implications in an algebraic sense. Existing IRs cannot efficiently support these operations.
- Lower the technical barriers for both crypto researchers and hardware engineers by decoupling front-end cryptographic algorithms from the back-end hardware platforms. The algorithms and solutions built on HLG framework can be written once and run everywhere. Researchers and engineers don't need to understand each other.
- Try to reduce the cost of data transfer between CPU and GPU/FPGA/dedicated hardware, by providing the intermediate representation (IR) of the computing graph for hardware compute engine, which allows task scheduling without help from CPU.

We have implemented CKKS algorithm based on HLG framework, together with a compute engine for multiple CPU cores. Experiment shows that we can outperform SEAL v3 Library in several use cases in multi-threading scenarios.

Keywords: Fully Homomorphic Encryption, Residue Number System, Computing Graph, Heterogenous Computing, Intermediate Representation

1 Introduction

Fully homomorphic encryption (FHE) used to be the holy grail of cryptography. By allowing computations on Ciphertext, FHE algorithm can lead to new paradigm of computing. When computations can happen in insecure environments like public cloud, novel solutions can be designed to support privacy protection and data processing at the same time, which seems to be impossible with conventional encryption schemes.

Since Gentry et al. proposed the first generation of FHE [18] in 2009, many improved schemes such as BFV [6,16] and BGV [7] have been proposed as leveled designs which are called second-generation FHE algorithms. In these schemes, in order to control the propagation of noise, polynomials with multi-precision coefficients are used. Bigger coefficients allow more multiplication levels. However the parameters have to been chosen after the required multiplication level is known. Besides that, bootstrapping operation can refresh noise level in a ciphertext and turn the somewhat homomorphic encryptions scheme above into a fully homomorphic encryption scheme.

In 2013, Gentry, Sahai and Waters [20] proposed a FHE scheme which avoids the complicated relinearization technique, with the following up works FHEW [15] and TFHE [11–13] which support fast bootstrapping for the evaluation of homomorphic Boolean gates are called third-generation. Chillotti et al. extended the TFHE scheme and proposed programmable bootstrapping techniques(PBS) [14] which enables the homomorphic evaluation of any function((including non-linear functions) of the message within 8-bit precision.

Many people have formed a consensus that FHE will be play a very important role for data security in the future. However, FHE is still suffering from the low performance. We believe that algorithm breakthrough and hardware acceleration are two key factors to improve FHE performance. In the next section, we will introduce the motivation for the design of HLG framework, especially challenges in both researches and implementations of FHE.

1.1 Our Contributions

In this paper, we propose a framework called Heterogenous Lattice Graph (HLG) for development and transformation of computing graphs for FHE. Inside HLG framework, we also designed a new IR system called HLG IR, whose data placeholders can be arbitrarily split and combined.

Basically, HLG framework aims for specific use cases of computing graphs with following features:

- Arbitrary splitting and combinations of data placeholders.
- Customizable operator expansion function: automatically removing higher-level operator and replace with one or multiple lower-level operators.
- Customizable granularity of operator expansion: user can define when the operator expansion ends by providing a set of primitive operators: this will allow the same front-end code to support different instruction sets on different hardware platforms.
- Does not support control flow at the graph level: computing graphs on ciphertexts won't contain any control flow.

The HLG framework is extendible. User can define customized data (operand) types and operators into this framework and develop any computing graphs for their specific domain.

We created operand types and operators for Ring Learning With Error (RLWE) related algebra, i.e., polynomials, including operators for algorithms in RNS system as extension plug-ins to HLG framework. Then based on these RLWE components, CKKS algorithm is implemented, including operand types of Plaintext, Ciphertext, RelinKey, GaloisKey and related operators such as Encode/Decode, KeyGen, Encrypt/Decrypt, Addition, Multiplication, Relinearization, Rotation and Bootstrapping etc. These operands and operators can be provided to developers to help them develop computing graphs for their own applications.

In addition, we built compute engines with support for multiple CPU cores. The computing graph generated by HLG framework can be loaded and executed by these compute engines. The compute engines support automatic parallel task scheduling and memory management to ensure that the computing powers of CPU cores are fully used.

We have compared performance of both HLG and SEAL in different scenarios such as MNIST digit recognition, calculation of square root etc. According to our benchmark results, HLG-CKKS library can outperform SEAL v3 library in multi-thread executions. Details can be found in section 6.

1.2 Organization of this paper

This paper is organized as follows.

- Section 2: Describes the design rationale of HLG framework, especially the challenges HLG intended to solve in both research and implementation.
- Section 3: Details of the components in HLG framework are introduced in this section.
- Section 4: if the reader just want to learn how to use HLG-CKKS library, he can skip top this section to check the operand/operator types and other user interfaces provided to cryptographers and developers.
- Section 5: details of implementation and optimization of the underlying primitive operators in HLG are introduced.
- Section 6: Details benchmark results of HLG with a comparison to SEAL library is introduced in this section. We also introduce an interesting observation of the performance bottleneck with multiple CPU cores.
- Section 7: we conclude this paper and look to the future plan in the last section.

2 Design Rationale of HLG

HLG is designed to address the challenges in both researches and implementations of FHE.

2.1 Challenges in researches of FHE

2.1.1 Deep technology stack for FHE

When talking about "researches" of FHE, we could refer to different levels:

- Design of application solution for FHE. Hybrid solution with FHE, MPC and more.
- Domain-specific application-level compiler.
- Design and improvement of FHE algorithm. Such as boosted key switch and improved bootstrapping algorithm.
- Design and improvement of algorithms at algebraic level. Such as RNS algorithms in double CRT form.
- Software framework for development.
- Implementation on specific hardware platform, i.e. GPU, FPGA etc.
- Hardware-level graph optimization and compiler.
- Dedicated hardware designed for accelerating FHE.

This is a deep technology stack.

If the goal of research is to publish papers only, working on single layer in the stack is enough. However, in order to promote practical applications for different use cases, we have to consider multiple layers or even all layers at the same time. This is the reason why research of FHE is extremely hard: all layers in this stack are tightly coupled together and formed a complex system. In order to achieve practical performance, one has to consider algorithm, parameter selection, application solution and hardware acceleration. Very few people can have expertise in multiple layers at the same time, which means progress in extending practical use cases of FHE would be very slow. In our opinion, this is the primary obstacle to large-scale adoption of homomorphic encryption.

2.1.2 Need for decoupling of technology stack for FHE

In order to promote the rapid progress of FHE research, decoupling is necessary. Imagine if cryptographers and hardware engineers can work independently in their respective areas of expertise and somehow their work can be magically integrated together with little effort, then experts in a single domain can efficiently participate in the collaboration on research of FHE without need to understand how things work in other domains.

To achieve this goal of decoupling, we can refer to the technology stack of AI: we may need a framework like Tensorflow [3] or PyTorch [27]. With similar framework as development tool, cryptographers can focus on the algorithm level and implement new algorithms and solutions without considering where these code would be executed and how to make it efficient. On the other hand, hardware engineers do not have to understand how algebraic algorithms in the RNS system work and how does bootstrapping works. They only need to consider how to finish the computations of given computing tasks as fast as possible.

In the project of openFHE [4], there is already a design for decoupling. The Hardware Abstraction Layer (HAL) is used to switch between different back ends for implementation of primitive operators for polynomials. With this HAL layer, developers can write code without considering optimization for specific hardware platform. However, there is limitation in the framework of openFHE: Although it allows user to switch between hardware platforms, it is assumed that the task scheduling is done by CPU. Then it is still necessary to transfer data between CPU and heterogenous hardware, which would be the performance bottleneck. Besides, the granularity of primitive operators is fixed and not easy to change. The flexibility to change level of primitive operators might be necessary if we need to allow the same front-end code to run on different types of hardware in the future.

2.2 Challenges in Implementations of FHE

2.2.1 RNS system for multi-precision arithmetic

Trivial implementation of operations on multi-precision polynomials are slow. For example, multiplication of multi-precision integers has time complexity of $O(l^2)$, where l is the bit length of an integer. By introducing Residue Number System (RNS), which is based on Chinese Remainder Theorem (CRT), modular addition and multiplication of multi-precision polynomial are equivalently transformed into parallel modular operations on single-precision polynomials. The complexity of both addition and multiplication becomes $O(l)$. Furthermore, by introducing Number Theoretic Transform (NTT) with complexity of $O(n\log(n))$, where n is degree of the polynomial, addition and multiplication of single-precision polynomial are transformed into element-wise operations. Thus the complexity has been reduced to $O(n)$, if we keep the polynomials in NTT form. After putting polynomial with multi-precision coefficients in CRT form (type name CRTPoly in hlg) into NTT transformation, we will get output polynomial in double-CRT (CRT+NTT) form (type name CRTPolyNTT in hlg), which is necessary in state-of-the-art high-performance implementations of FHE algorithms.

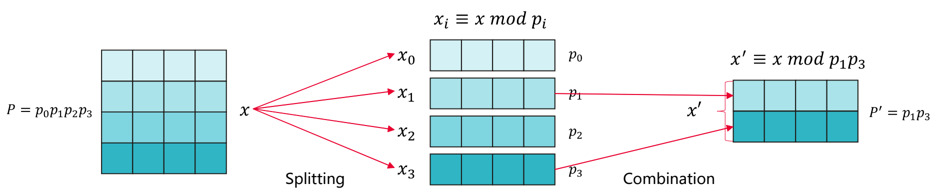


Figure 1: Algebraic implication of data splitting and combination in RNS system

In an RNS system, the operations of data splitting and combination have algebraic implications, as shown in Fig. 1:

1. Split multi-precision coefficient into single-precision coefficients: stands for modulo operation regarding single-precision modulus.
2. Combine multiple single-precision coefficients into one: stands for change of RNS base where the target base is a subset of the original base.

In addition, in the implementation of many algebraic operators, data splitting and combinations are also required. For example: Rescale [9], Fast Basis Conversion [5], Boosted Key Switch [19, 21, 22] and many other operators in RNS algorithms.

2.2.2 Hardware acceleration is necessary for FHE

Computation on ciphertext is slow even with state-of-the-art FHE algorithms: it is more than dozens of thousand times slower than computation on plaintext, on average. A breakthrough on the algorithm level is becoming more and more challenging. There is a growing consensus in both academia and industry that hardware acceleration is necessary in order to allow FHE to reach practical performance and be widely adopted by real applications.

In 2020 [1], DARPA has published a call for proposal for project DPRIVE. This project is aiming at creating novel technology stacks dedicated for FHE from scratch: hardware, compiler and tools for development.

2.2.3 Data transfer is the main performance bottleneck for FHE

However, hardware implementation of FHE is also difficult: existing hardware platforms are not quite suitable for FHE. Almost all existing hardware architectures cannot process high dimensional NTT transformation very efficiently, due to its all-to-all data access pattern. Data transfer between CPU and heterogenous hardware takes more than 80% of end-to-end time consumption, which is the main bottleneck.

2.2.4 Need for Dedicated Intermediate Representation

There could be multiple ways to reduce cost of data transfer in heterogenous computing scenarios. One possible way is to leave the burden of task scheduling to the hardware itself, without help form CPUs. Then the data transfer between CPU and the hardware can be greatly reduced. In order to instruct the hardware on how to schedule the tasks, an Intermediate Representation of the computing graph is required.

However, existing IR systems (such as LLVM IR [24], MLIR [25]) cannot support splitting and combination of data placeholders natively. In these traditional IR system, data placeholders are always defined and used as a whole piece throughout its life cycle.[1] For RNS system, we need a brand new IR, which supports splitting and combinations of placeholders in a natural way.

3 Description of HLG framework

3.1 Components and work flow of HLG framework

There are four components in the library of HLG-CKKS now:

[1] Note that there may be alternative solution to express computing graphs in RNS as a dialect of MLIR. However, HLG is an independent exploration. When the paper of MLIR is published, we have already written a lot of code. We have to finish the code in order to verify if the idea works. In the future, we will consider to migrate to MLIR in order to reuse existing tool chain.

- **HLG framework**: including definition of raw data placeholder, base class of operand and operator, pre-defined templates for development of customized derived operand classes, etc. Also, it provides the Context which is used throughout the process of defining and processing the computing graph. Details of the Context will be introduced in section 3.7
- **Extension plugins**: Custom operand types and operators, such as CRTPoly(multi-precision), RNSPoly(single-precision) and related operators like NTT, BasisExtension etc. at RLWE level. operand types Plaintext, Ciphertext, RelinKey, GaloisKey and operators like Encode/Decode, KeyGen, Encrypt/Decript, Multiplication, Relinearization, Rotation etc. at CKKS algorithm level.
- **Work builder**: transforming original computing graph with high-level operand types and operators into a "work", which is an optimized (such as reduplicated and pruned) and simplified graph of tasks in the form of HLG IR.
- **Compute engine on CPUs**: universal execution engine which can load and execute works in the form of HLG IR, with support for automatic task scheduling on multiple cores and memory management.

The work flow of HLG framework is as follows:

1. Build original high-level computing graph with pre-defined or customized operand types and operators, with help from basic functionalities from the framework itself.
2. Use work builder to expand higher-level operators into lower-level operators by calling the expansion function defined for each operator, until only primitive operators left. During this process of operator expansion, the placeholders might be split into pieces too.
3. Transform and generate simplified computing graph with primitive operators only as the "work", which can stay in memory in the form of an object of protobuf [32] data type "ProtoWork", or be serialized as a binary file.
4. Automatic pre-computation and pruning of the original work. If all input operands of an operator are constant or pre-computable, this operator can be pre-computed and its output operand is pre-computable.
5. Load the work into a compute engine for initialization. Run the compute engine providing input data and the engine will output result data after the execution ends.

The components in HLG framework and workflow is shown in Fig. 2.

Note that after initialization, the engine can run with different input data for multiple times. The user can also choose which compute engine to use in steps 5. If we choose a single-thread CPU engine, this work will be executed with one thread. If we choose a multi-thread CPU engine, this work will be executed with multiple threads and automatic task scheduling strategy.

For example: HLG library has provided two multi-thread CPU engines: MyEngine and LockFreeEngine. MyEngine is implemented with mutex and LockFreeEngine is implemented with lock-free queue. They have different strategies for task scheduling and memory management. Their performance may vary for different types of works, but their difference remain within a limited range.

We also plan to support compute engines for GPU, FPGA and other platforms in the future. For these compute engines on heterogenous platforms, we don't need to change the code written in step 1 above. However, we may change the parameters a little bit for the work builder in step 2. Hardware-specific compiling and graph optimization might also be included between step 2 and 3.

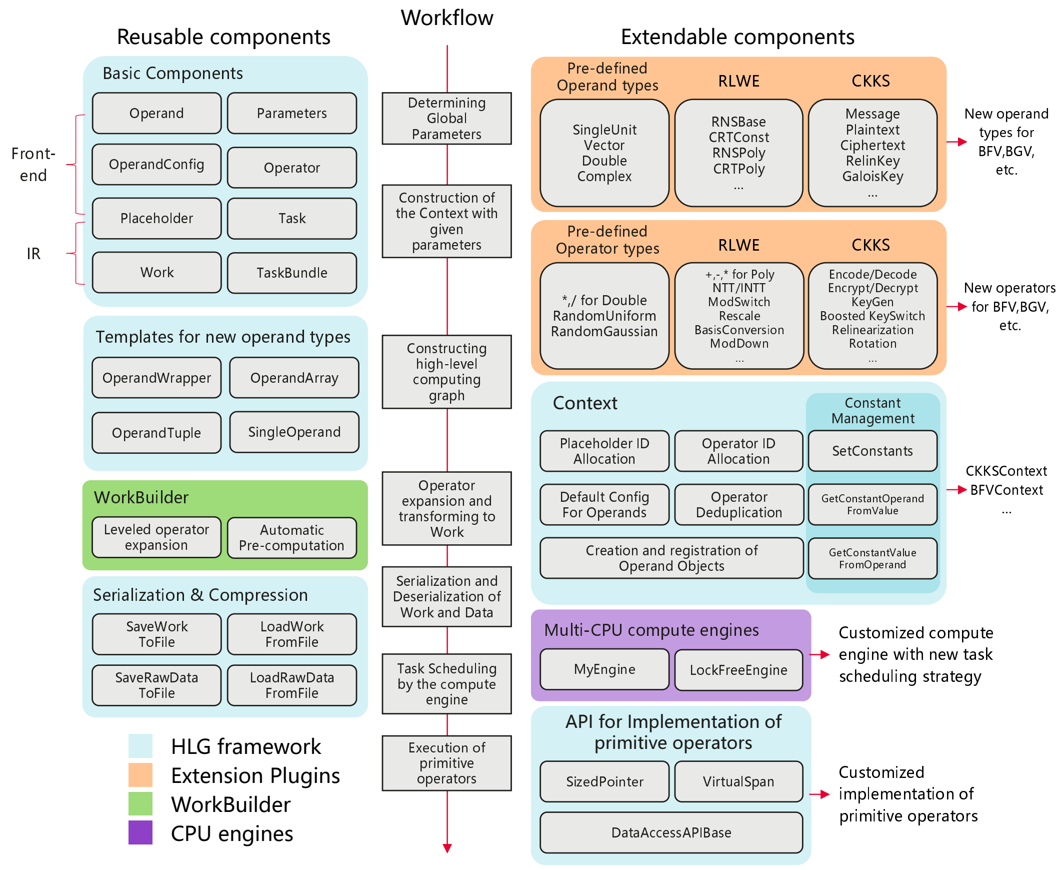

Reusable components
Workflow
Extendable components
Basic Components
Operand
Parameters
OperandConfig
Operator
Placeholder
Task
Work
TaskBundle
Front-end
IR
Templates for new operand types
OperandWrapper
OperandArray
OperandTuple
SingleOperand
WorkBuilder
Leveled operator expansion
Automatic Pre-computation
Serialization & Compression
SaveWork ToFile
LoadWork FromFile
SaveRawData ToFile
LoadRawData FromFile
HLG framework
Extension Plugins
WorkBuilder
CPU engines
Determining Global Parameters
Construction of the Context with given parameters
Constructing high-level computing graph
Operator expansion and transforming to Work
Serialization and Deserialization of Work and Data
Task Scheduling by the compute engine
Execution of primitive operators
Pre-defined Operand types
RLWE
CKKS
SingleUnit Vector Double Complex
RNSBase CRTConst RNSPoly CRTPoly ...
Message Plaintext Ciphertext RelinKey GaloisKey ...
New operand types for BFV,BGV, etc.
Pre-defined Operator types
*,/ for Double RandomUniform RandomGaussian
+,-,* for Poly NTT/INTT ModSwitch Rescale BasisConversion ModDown ...
Encode/Decode Encrypt/Decrypt KeyGen Boosted KeySwitch Relinearization Rotation ...
New operators for BFV,BGV, etc.
Context
Constant Management
Placeholder ID Allocation
Operator ID Allocation
SetConstants
Default Config For Operands
Operator Deduplication
GetConstantOperand FromValue
Creation and registration of Operand Objects
GetConstantValue FromOperand
CKKSContext BFVContext ...
Multi-CPU compute engines
MyEngine
LockFreeEngine
Customized compute engine with new task scheduling strategy
API for Implementation of primitive operators
SizedPointer
VirtualSpan
DataAccessAPIBase
Customized implementation of primitive operators

Figure 2: Overview of components in HLG framework and related workflow

3.2 Placeholder and operand Types: data split and combination

3.2.1 Definition of Placeholder representing raw data

The core design goal of HLG IR is to support arbitrary splitting and combination of placeholders. We assume there is a virtual memory space as a sequence of unit cells with infinite length, where each cell stands for a data unit with fixed length. The placeholder can be defined as a continuous segment in this sequence. Name each unit cell as an integer index number, starting from 0 to infinite. Then one placeholder can be written in 3 equivalent forms: range form, set form and normal form.

- Range Form: $[left, right)$
- Set Form: $\{left, left + 1, left + 2, ..., right - 1\}$
- Normal Form: $Ph\{ptr = left, size = right - left\}$

For example: the placeholder $[2, 7) = \{2, 3, 4, 5, 6\} = ph\{ptr = 2, size = 5\}$ includes 5 unit cells at positions from index 2 to 6. With this definition, splitting placeholder become simple: $[2, 7)$ can be split into smaller segments $\{[2, 3), [3, 5), [5, 7)\}$.

There are several notes about placeholders:

- **Note 1**: It is not necessary to specified the bit size of each unit cell in the placeholder during the process of creation and transformation of the computing graph. When we want to execute the computing graph, we can specify the size of one unit cell. For example: if we want each unit cell to be 64 bits = 8 bytes, we can allocate $5 \times 8 = 40$ bytes of memory for the placeholder $[2, 7)$ above.

- **Note 2**: The HLG framework does not recognize types of data represented by placeholders. Only when these raw data enters the implementation of primitive operators, these operators can recognize the data types.
- **Note 3**: The same index of unit cell in the placeholder indicates the same real memory space. The memory allocated for placeholders without overlapped unit cells would be different. However, the memory allocated for adjacent placeholders in the virtual space may not be adjacent in real memory.
- **Note 4**: In HLG framework, placeholders are allocated in a continuous way in the virtual memory space. However, some intermediate placeholders would be discarded during creation and transformation of the computing graph. So, in the final computing graph, the complete placeholder set may not be continuous.

3.2.2 Operand as typed placeholder with meta data

Using placeholder directly is hard and error-prone. In order to improve usability of placeholder, HLG framework support user-defined operand types. First, each operand type should have a default placeholder as the raw data. Second, it can also have meta data to help define data type and specify how to split and combine the placeholder it represents.

For example: operand type CRTPoly has a placeholder that stores the coefficients of the polynomial and also another member of type RNSBase:

$$CRTPoly\{placeholder = [3, 3L + 3), rns\_base = RNSBase\{0, 1, 2\}\}$$

where L is polynomial degree and the RNS base does not represents the raw data of the CRTPoly. However, it can be used as meta data in at least two use cases:

1. When we want to split one CRTPoly into multiple child RNSPolys, the meta data can provide enough information on how it should work. The CRTPoly above can be split into:

$$RNSPoly1\{placeholder = [3, L + 3), modulus = SingleUnit\{0\}\}$$
$$RNSPoly2\{placeholder = [L + 3, 2L + 3), modulus = SingleUnit\{1\}\}$$
$$RNSPoly3\{placeholder = [2L + 3, 3L + 3), modulus = SingleUnit\{2\}\}$$

   We call these RNSPolys as child operands of the parent operand CRTPoly here.

2. When we call NTT operator on RNSPoly1, its member modulus can tell operator NTT to choose from the pre-computed tables of twiddle factor for the given modulus stored in placeholder $\{0\}$. So the interface of NTT operator can be as simple as $NttRns(RNSPoly1)$ without the modulus as the input parameter as well.

3.2.3 Birth parent, step parent and virtual placeholder

In the example of section3.2.2, the placeholder of the CRTPoly is split into small pieces and each child RNSPoly has inherit one of them. We call CRTPoly the "**birth parent**" of these child RNSPolys.

We can combine RNSPoly1 and RNSPoly3 to form a new CRTPoly2 with smaller RNSBase:

$$CRTPoly2\{placeholder = virtual, rns_base = RNSBase\{0, 2\},$$
$$children = \{RNSPoly1, RNSPoly3\}\}$$

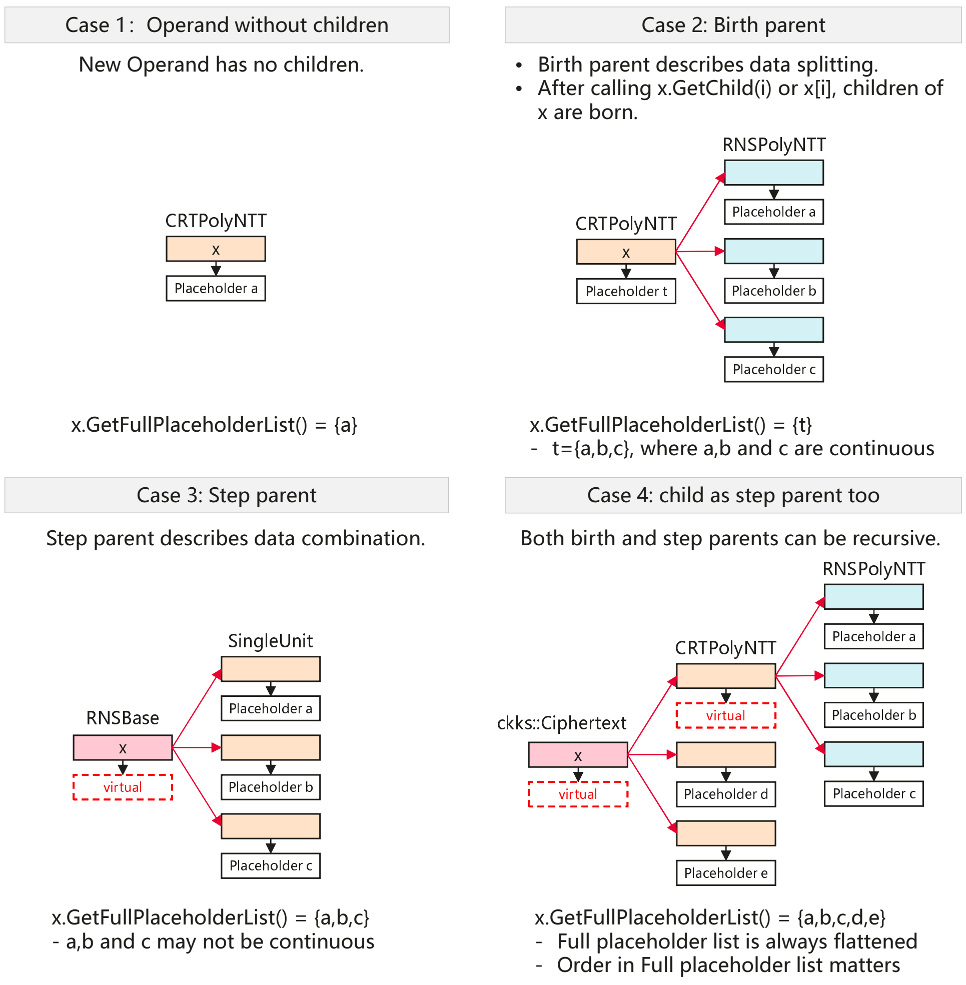

Case 1 : Operand without children
New Operand has no children.
CRTPolyNTT
x
Placeholder a
x.GetFullPlaceholderList() = {a}
Case 2: Birth parent
• Birth parent describes data splitting.
• After calling x.GetChild(i) or x[i], children of x are born.
RNSPolyNTT
CRTPolyNTT
x
Placeholder t
Placeholder a
Placeholder b
Placeholder c
x.GetFullPlaceholderList() = {t}
- t={a,b,c}, where a,b and c are continuous
Case 3: Step parent
Step parent describes data combination.
SingleUnit
RNSBase
x
virtual
Placeholder a
Placeholder b
Placeholder c
x.GetFullPlaceholderList() = {a,b,c}
- a,b and c may not be continuous
Case 4: child as step parent too
Both birth and step parents can be recursive.
RNSPolyNTT
CRTPolyNTT
ckks::Ciphertext
x
virtual
virtual
Placeholder a
Placeholder b
Placeholder c
Placeholder d
Placeholder e
x.GetFullPlaceholderList() = {a,b,c,d,e}
- Full placeholder list is always flattened
- Order in Full placeholder list matters

Figure 3: Operand as birth parent and step parent

Here CRTPoly2's placeholder has to be **virtual**, which means it does not have its own original placeholders. The placeholder list of CRTPoly2 are determined by its children, i.e. RNSPoly1 and RNSPoly3. In this case, we call CRTPoly2 the "**step parent**" of RNSPoly1 and RNSPoly3. Refer to Fig. 3 as examples of birth parent and step parent.

In order to allow users to check the real placeholder list, we have provided a method "GetFullPlaceholderList()" for the operand types, which will collect all non-virtual placeholders from the operands themselves and their children recursively. Here we have several examples:

$$\begin{aligned} CRTPoly.GetFullPlaceholderList() &= \{[3, 3L+3)\} \\ RNSPoly1.GetFullPlaceholderList() &= \{[3, L+3)\} \\ RNSPoly2.GetFullPlaceholderList() &= \{[L+3, 2L+3)\} \\ RNSPoly3.GetFullPlaceholderList() &= \{[2L+3, 3L+3)\} \\ CRTPoly2.GetFullPlaceholderList() &= \{[3, L+3), [2L+3, 3L+3)\} \end{aligned}$$

Note that the order of placeholders in the full placeholder list matters, which defines the order how the fragmented placeholder/data pieces are combined.

Here CRTPoly2 cannot have its own placeholder since it is a step parent and its placeholder should be defined by its children. Since the placeholders of its children are not continuous, we set step parent's placeholder to virtual to avoid inconsistency. CRTPoly can have its placeholder as long as it is consistent with the sub placeholders of its child RNSPolys. This can help us avoid excessive fragmentation of placeholders and reduce memory consumption for representation of the computing graph.

However, sometimes the placeholder of some child RNSPoly could be changed and it would no longer be consistent with its birth parent. Then we would set the placeholder of its birth parent to virtual, in order to resolve the inconsistency. We refer this to the "Absorb" operation in section 3.2.5.

3.2.4 Raw data size of an operand

We define the raw data size of an operand as the sum of the sizes of all placeholders in its full placeholder list. In the examples defined in section 3.2.3:

$$\begin{aligned}
RNSPoly1.RawDataSize() &= Size([3, L+3)) = L \\
CRTPoly.RawDataSize() &= Size([3, 3L+3)) = 3L \\
CRTPoly2.RawDataSize() &= Size([3, L+3)) + Size([2L+3, 3L+3)) = 2L
\end{aligned}$$

3.2.5 The Absorb operation for operands

HLG framework supports a useful operation called "Absorb". It is used to merge two different operand objects' placeholders into one. The two operand objects should have the same type and raw data size.

The operation of operand a "absorbs" operand b can be written in several equivalent forms:

$$\begin{aligned}
&a <<= b; \\
&a.Absorb(b); \\
&Absorb(a, b);
\end{aligned}$$

For example: Suppose that $a = c + d$ and $b = NTT(e)$ are the computing graphs we have created. After calling "$a <<= b$": 1) Now both a and b refers to the same operand object. The original operand object a has disappeared unless it has other references store somewhere else. 2) The placeholder of b is discarded and replaced by the placeholder of a, which means, in the execution of this operator NTT, the output data will be stored in the memory allocated for a's placeholder, instead of b's original placeholder.

The absorb operation is useful in the implementation of operator expansion, which will be introduced in details in section 3.3.4.

3.2.6 Delayed pointer synchronization and PtrSyncer<O> type

This functionality is used to implement the "Absorb" operation defined in the last section. From the definition of Absorb operation, we know that the goal is to make two operand objects merge as one. However, there is a major problem that both operand a and b might have multiple references somewhere else. But in the Absorb function, we only have "local" perspective. There is no way we can get the "global" view and figure out where and how many these references to a and b are. In order to solve this problem, we introduced two features: 1) registration of operand objects 2) delayed pointer synchronization.

Every time a new operand object is created, we store it in a map inside the Context object with the operand's hash value[2] as the key in the map. When we need to store a pointer of an operand object, we use the PtrSyncer<O> instead.

[2]The definition of the have function of an Operand is introduced in section 3.3.3

The PtrSyncer<O> type has a smart pointer to object of operand type O and also a hash value. Every time we call its Ptr() method, it will try to check if the hash value stored in PtrSyncer<O> object is the same as the hash value in the operand object. If the hash values are different, PtrSyncer<O> object will find the correct operand object in the operand map in the Context object, using its hash value as the key. Then update its smart pointer to the correct object. If the hash values are consistent, it just returns the smart pointer directly.

We call this "delayed pointer synchronization", since the synchronization does not happen when we change the hash value of the operand object. The algorithm is described in Alg. 1. It is only when we call the Ptr() method of PtrSyncer<O>, the synchronization will happen. Then in the implementation of Absorb operation, we only need to set the hash value of some operand object a to hash value of b, which is still an possible operation with "local" perspective only. After that, when we try to get the pointer to operand object a again, the PtrSyncer<O> will give us the pointer to operand object b instead. Then the problem of implementing Absorb operation is solved.

Algorithm 1 Delayed pointer synchronization

Assumption 1: *ObjMap* is a global map which stores all objects of operands.
Assumption 2: PtrSyncer object has two members: Operand Ptr and Hash.
Input: PtrSyncer *ps*. Triggered by calling *PtrSyncer.GetPtr*() interface.
Output: Synchronized pointer to correct operand object

```
1: if $ps.Hash \neq ps.Ptr.Obj.Hash$ then
2:     $p \leftarrow ObjMap[ps.Ptr.Obj.Hash]$
3:     $ps.Ptr \leftarrow p$
4:     $ps.Hash \leftarrow p.Obj.Hash$
5: end if
6: return $ps.Ptr$
```

Finally, we have a working algorithm for the Absorb operation. The pseudo code is as follows:

Algorithm 2 Absorb operation: *src* <<= *tar*, or *src* absorbs *tar*

Input 1: Operand src, the source operand
Input 2: Operand tar, the target operand

```
 1: $ph \leftarrow src.Placeholder$
 2: $parent \leftarrow src.BirthParent$
 3: for $i = 0$ to $tar.Children.Size$ do              ▷ will do nothing if no children exist
 4:     $src.Children[i] <<= tar.Children[i]$       ▷ recursively absorb each of the children
 5: end for
 6: $ch \leftarrow src.Children$
 7: $src.Hash \leftarrow tar.Hash$   ▷ $src$ and $tar$ will be merged as one by triggering PtrSyncer
 8: $tar.Placeholder \leftarrow ph$
 9: $tar.Children \leftarrow ch$
10: $p \leftarrow tar.BirthParent$
11: while $p$ is not null AND $p.Placeholder$ is not virtual do
12:     $p.Placeholder \leftarrow virtual$        ▷ recursively setting all ancestors of $tar$ to virtual
13:     $p \leftarrow p.BirthParent$
14: end while
15: $tar.BirthParent \leftarrow parent$                              ▷ fixing the birth parent
```

3.2.7 Template OperandWrapper<O>

Since we have registration of operand objects in a map of Context object, we should not store the pointer of the operand object in anywhere else. In order to allow users to use PtrSyncer<O> type easily, we introduced template OperandWrapper<O> with an internal pointer of PtrSyncer<O>, which provides many useful functions such as operator-> and allow user to use OperandWrapper like a raw pointer to the operand object. The wrapper object can be stored on the stack, while all operand object should stay in the map in the Context object.

In HLG, all operand types (derived from class Operand) are named as "XXXImpl", whose wrapper type are defined as "XXX". For example, we can define customized Operand type and its wrapper type as:

```
class CiphertextImpl: public Operand;
class Ciphertext: public OperandWrapper<CiphertextImpl>;
```

Only the class Ciphertext is provided to users at the application level. The object of CiphertextImpl can only be created using constructors of Ciphertext.

3.2.8 Pre-defined templates for implementation of customized operand types

HLG framework provides several pre-defined templates to help user define their own operand types:

1. **OperandArray<O>**: Define a new operand type as an "array" of operand type O, where user can specify the array size. In an OperandArray<O> object, each child operand (of type O) should have the same raw data size. This is similar to std::vector<X> in c++.

2. **OperandTuple<O,P,Q...>**: Define a new operant type as a tuple of O, P, Q... (variadic template), where type and raw data size of each child operand can be different. This is similar to std::tuple<X,Y,Z> in c++.

3. **SingleOperand<O>**: A specialized derived class of OperandArray<O> with fixed array size of 1, which means it will have only 1 child operand of type O.

Note that object of type SingleOperand<O> and its only child will have the same full placeholder list. They are actually referring to the same placeholder, i.e. data. They just provide us different ways of looking at the same piece of placeholder. The template SingleOperand<O> is supposed to be used to define "inheritance" between operand types in HLG framework, where original inheritance of c++ does not work correctly here[3].

3.3 Operator Type

An operator has the following members:

- Type of operator defined as int type or enum type.
- ID of the operator as distinct integer number with type uint64_t.
- The input operands of the operator. The number of inputs can be zero.

Note that the operator type does not have a member as the output operand.

In HLG framework, in order to allow multiple equivalent copies of the output operand, there is an assumption that each operator should have only one output operand. We give

[3]Original class inheritance of c++ will fail when delayed pointer synchronization (defined in section 3.2.6) is triggered. However, inheritance using SingleOperand<O> can avoid this problem.

the operand type a member called SourceOp(), which is a pointer to its source operator which indicates that this operand is the output of this operator.

The other assumption of HLG is that all output operands of the operators stand for real data, which we need to allocate memory for. The input operands of the operators are just full or partial reference to some other output operands, which we don't need to allocate memory for.

In the definition of each operator, we need specification of the following functions:

1. Operator creation function: creates new operators with given input operands, the output operand of this new operator will also be new. This is provided to the developers.

2. Operator expansion function: defines how a higher-level operator will be expanded to how-level operators. Crypto researchers or developers at the algorithm/algebra level This is provided to the work builder for auto expansion of operators.

3.3.1 Categories of operators

There are several different categories of operators:

1. **Normal operator**: stands for normal computational operator, which will be either expanded into lower-level operators or be implemented as the primitive operator.

2. **INPUT operator**: if output->SourceOp() is an INPUT operator, this output operand is one of the input parameters of the whole computing graph(work), where the real data provided to Engine::Run(...) interface will be assigned to the memory allocated for these operand with INPUT operators. We will talk about design of the execution engines in section 3.10.

3. **CONSTANT operator**: the original constant values and the pre-computed values will be stored in the memory for these operands with SourceOp() equals CONSTANT operator. Constant operator will trigger automatic pre-computation in hlg framework, which we will talk about in section 3.6.

4. **RANDOM operator**: A special category of operator, which is not pre-computable and cannot be deduplicated. There are only two types of primitive random operators: $RANDOM\_UNIFORM$ and $RANDOM\_GAUSSIAN$ now. For more complicated random operator, user can define their own new operator based on these primitive ones.

3.3.2 Creation and deduplication of operators

HLG framework provides a function MakeOperator, which is the unified interface for creating operators. The interface looks like:

$$MakeOperator(Context, OP\_TYPE, output, input_0, input_1, input_2, ...)$$

After calling this interface, a new operator of type OP_TYPE will be created, with the given input operands $\{input_i\}$. The pointer output->SourceOp() will also point to this new operator.

Note that this MakeOperator function supports **deduplication of operators**. If we call MakeOperator with the same OP_TYPE and input operands twice, only one operator will be created, as they have perfectly the same content. Furthermore, if duplicated operators are detected, the output parameter will be overwritten as the output of existing operator. For example:

```
void TestDeduplication(){
    CRTPoly a,b; //CRTPoly a and b with default parameters
    CRTPoly d(a->RnsBase()); //create new CRTPoly with a's RnsBase
    CRTPoly e(a->RnsBase()); //create new CRTPoly with a's RnsBase

    // now d!=e, since they have different placeholders
    MakeOperator(a->Context(), ADD_CRT_POLY, d, a, b);
    MakeOperator(a->Context(), ADD_CRT_POLY, e, a, b);
    // now d==e, because of deduplication, e was overwritten as d
}
```

In order to implement this deduplication, we need to design an hash function of the operand.

3.3.3 Definition of hash function of an operand object

If an operand's SourceOp() is null or of type INPUT or CONSTANT, we define its hash value as

$$Hash(operand\_type, ph.ptr, ph.size)$$

where *ph* is the operand's placeholder. We can distinguish different operands with different placeholders with this definition of hash function.

In the other cases, the hash function of an operand is defined as:

$$Hash(operand\_type, OP\_TYPE, Hash(input_0), Hash(input_1), Hash(input_2), ...)$$

where $input_i = SourceOp().Input[i]$. This is a recursive definition, since the inputs of the source operator pare also operands. Note that the placeholder of the output operand is ignored here, which is necessary when we want two different output operands to have the same hash values as long as they have the same source operator type and input operands.

3.3.4 Levels of Operators and expansion of operators

As we mentioned in section 3.1, in step 2 of the work flow of HLG, the work builder will "expand" higher-level operators to lower-level operators. This process can be designed to be executed layer by layer, then each layer becomes simpler, as shown in Fig. 4. Here we explain how it works in more details.

Recall that we have used operator $NttRns(RNSPoly)$ as an example in section 3.2.2. Based on this operator $NttRns$ at the level of RNSPoly, we can define another NTT operator at the level of CRTPoly: $NttCrt(CRTPoly)$. The developer only need to write code with $NttCrt$ operator. HLG framework, together with the expansion function defined for $NttCrt$, will automatically remove this higher-level operator and replace with lower-level operators $NttRns$ on each of the child RNSPoly of the CRTPoly object.

Suppose we have $b = F(a)$ and we would like to expand operator F as $b = F(a) = H(G(a))$. Basically, we will create new intermediate operands as: $c = G(a)$ and $d = H(c)$. In HLG framework there is an assumption that when creating a new operator, a new placeholder will always be allocated for the output operand of this operator. As a result, operands d and b would have different placeholders. In case that operand d (or its children) has been used by other Operators as input, we need to update these references to placeholder of d for synchronization in order to avoid inconsistency. More specifically, we would like to keep the original placeholder of b, but as the output operand of operator H. We can achieve this by using the Absorb operation "$b <<= d$", or "$b <<= H(G(a))$" defined in section 3.2.5.

There is an example for operator expansion: operator ADD_MOD_CRTPOLY is split into multiple lower-level operators of type ADD_MOD_RNSPOLY. The number of generated lower-level operators here is determined by the CRTPoly's RNS size, i.e. the number of prime modulus number of children.

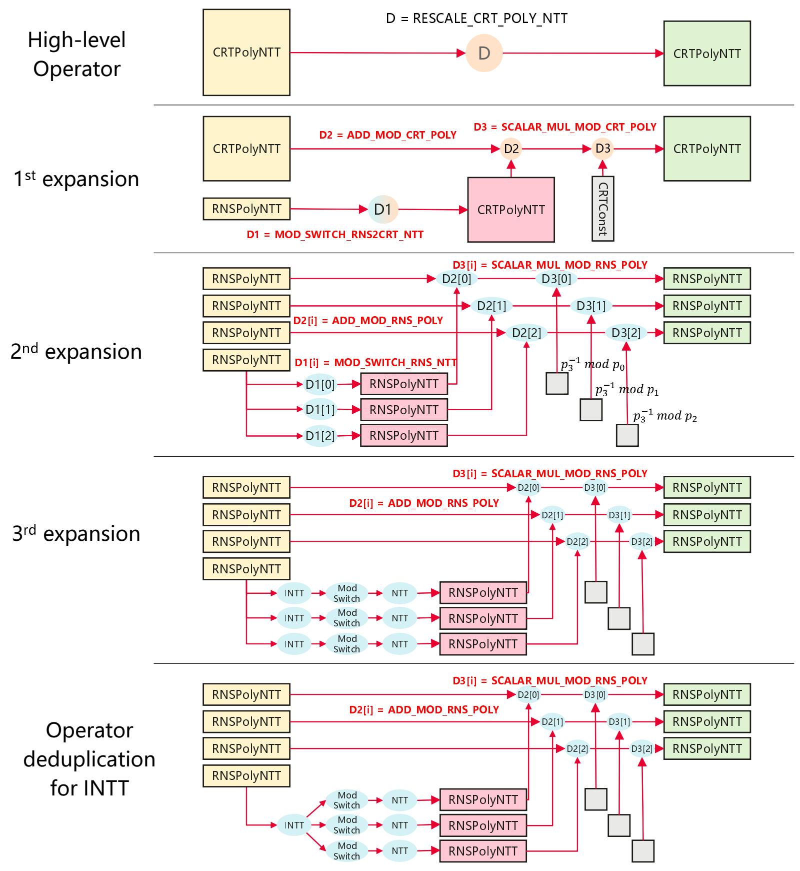


Figure 4: Leveled operator expansion in HLG

```
void ExpAddModCrtPoly(CRTPoly ret, CRTPoly a, CRTPoly b) {
    for (uint32_t i = 0; i < ret->RnsSize(); i++) {
        // a[i] and b[i] are RNSPolys, children of CRTPoly a and b.
        ret[i] <<= a[i] + b[i]; //ret[i] absorbs the sum of a[i] and b[i]
    }
}
```

- Note 1: the 1st input *ret* of the expansion function is the output operand of the original operator. The following inputs *a* and *b* are the input parameters of the original operator. In HLG framework, expansion function should be implemented with this format.

- Note 2: we can split the original output operand *ret* into pieces(children) and absorb different other operands of the same type.

3.3.5 Customization of granularity for operator expansion

In order to support switch between different granularity of primitive operators by just changing parameters for the work builder, we need some preparation work.

Table 1: Flexible-level operators

Operator	expansion function	implementation	note
OP_1	yes	no	high-level operator
OP_2	yes	yes	flexible-level operator
OP_3	no	yes	primitive-level operator

As shown in table 1:

- OP_1 is always a high-level operator which will be expanded every time. We don't need to define its implementation function.

- OP_3 is always a primitive-level operator which will not be expanded. We don't need to define its expansion function. Instead, it has to be implemented.

- OP_2 is a flexible-level operator, which has both expansion and implementation functions. If we set it as primitive operator, its implementation function will be called during execution. Otherwise its expansion function will be called.

For example, there is an algorithm called four-step NTT, which aims to improve performance of NTT operator in hardware platforms with limited cache size (or narrow bandwidth for data transfer). In the four-step ntt algorithm, operator NTT_{65536}(RNSPoly-level) is split into multiple operators of NTT_{256}(lower-level) and transposes. When we implement $NttRns$ on these platforms, it is not primitive and will be expanded. However when we need to implement it on mainstream CPUs, we want it to be primitive, since four-step NTT is slower than normal implementation of $NttRns$ on CPUs. $NttRns$ should be a flexible-level operator in this case.

Suppose that we have built an application based on HLG framework which has been tested on CPUs. Not a single line of code needs to be changed when we want to migrate it to other hardware platforms. What we need to do is to change the parameters of the work builder at the front end. We also need another compute engine on the target hardware platform, however these work can be independent with the code at the front end. Thus we can achieve decoupling of application-level code with hardware platforms, which can greatly reduce the effort of implementation and experiment for designing and applications of FHE.

3.3.6 Recovery of dependency between operators by checking overlapping of placeholder after expansion of operators

As we introduced in section 1.1, HLG is designed to support the following features at the same time: 1) arbitrary splitting and combination of placeholders 2) customizable granularity of operator expansion. In section 3.2.3, we also mentioned about the need to keep tracking parent and children of an operand, which would be convenient for the developers. Then we found some contradictions between these requirements.

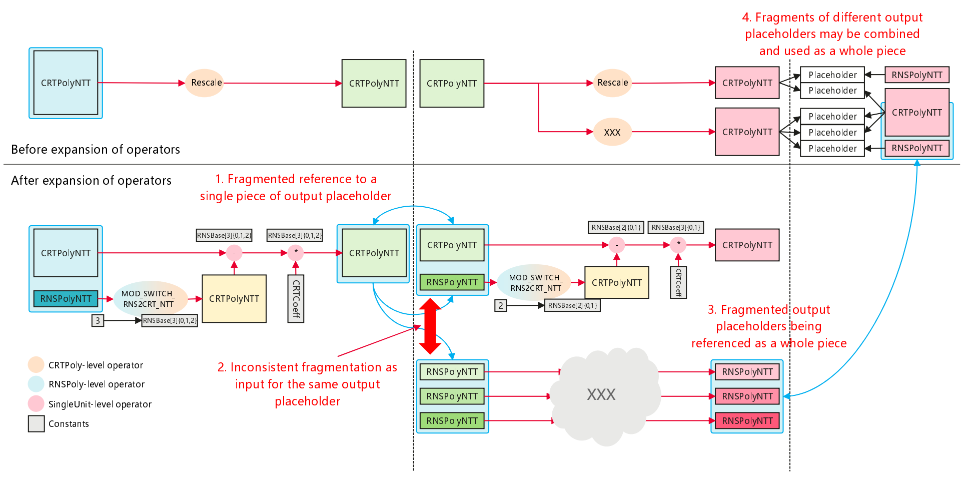

4. Fragments of different output placeholders may be combined and used as a whole piece
Before expansion of operators
After expansion of operators
1. Fragmented reference to a single piece of output placeholder
2. Inconsistent fragmentation as input for the same output placeholder
3. Fragmented output placeholders being referenced as a whole piece
CRTPoly-level operator
RNSPoly-level operator
SingleUnit-level operator
Constants
CRTPolyNTT
RNSPolyNTT
Rescale
XXX
Placeholder
MOD_SWITCH_RNS2CRT_NTT
CRTCoeff

Figure 5: Corner cases where parent/child relationship is not enough

In Fig. 5, we described several corner cases where the same output placeholder(s) are used with different granularity of fragmentation when referred by different operators as input. These inconsistent fragmentation is beyond the expressive power of simple parent-child relationships. The only choice we found is to give up tracking dependency between operators during the process of operator expansion. After the expansion is finished, we can recovery the dependency between the final set of primitive operators, by checking the overlapping of their input and output placeholders: if input placeholder of operator B is overlapping with output placeholder of operator A, we know that **operator B depends on operator A** and operator A should be executed before operator B.

The overlapping of placeholders can be defined as whether the intersection of the placeholders (as a set of unit cells) is not empty, which is very intuitive. However, for better performance and less memory requirement, we need to figure out how to find overlapping placeholder in the range form efficiently. Range form of placeholder is defined in section 3.2.1.

The idea is to use binary search. First, we collect the set of all output placeholders of all the primitive operators and sort them in ascending order. Then, for each of the input placeholder, we can have a quick algorithm to find ALL overlapping output placeholders in the sorted list. The algorithm is described in Alg. 3.

Algorithm 3 Find all overlapping placeholders of target placeholder ph in a sorted list L

Input 1: Target placeholder $ph = [l, r)$
Input 2: Sorted List L of Placeholders in ascending order, where $|L| = n$ and $L_i = [l_i, r_i)$ is the i-th placeholder in L
Output: Sub-list $L' = \{L_i \in L | L_i \text{ is overlapping with } ph\}$
Complexity: $O(log(n))$

```
1: Find the smallest index $min$ such that $r_{min} > l$        ▷ binary search, $O(log(n))$
2: Find the smallest index $max$ such that $l_{max} \geq r$     ▷ binary search, $O(log(n))$
3: $L' \leftarrow \{L_t \in L | min \leq t < max\}$              ▷ $L'$ can be empty
4: return $L'$
```

Note that the placeholders in the sorted list L will not overlap with each other, or the assumption of HLG framework will be broken: all operators should have distinct output placeholders.

3.4 Task, Work and WorkBuilder

After the expansion of operators, we have a list of primitive operators only, which is an equivalent form of the original computing graph. Now we need to define a compact format to store the graph. Here we introduce several concepts:

- **Task**: A compact form of an operator without any unnecessary meta data, which has vector<vector<PlaceholderPtr>> as input and vector<PlaceholderPtr> as output.

  Note1: The ID of a task is the same with the ID of the original operator. Task is the compact form of primitive operator.

  Note2: Here vector<PlaceholderPtr> is the compact form of an operand. It can be extracted using Operand::GetFullPlaceholderList() interface. If the placeholder of an operand is a whole piece, size of the vector<PlaceholderPtr> would be 1.

- **TaskBundle**: A compact form defining a computing graph with multiple tasks, including: 1) A full set of all tasks 2) dependency relationship between the tasks 3) list of initial tasks and other tasks 4) sorted list of all output placeholders of all tasks 5) map of each placeholder's parent placeholder.

  Note: Data of items 2) to 5) of TaskBundle above can be extracted from the full set of tasks. We store these auxiliary data here for the convenience of the compute engines, since these data would be useful for task scheduling and memory management.

- **Work**: A work includes: 1) a TaskBundle 2) input and output placeholders of the work 3) constant value pool for the const placeholders. A work has all required information to define a complete computation work.

- **WorkBuilder**: The tool to transform the original computing graph with operands and operators into a Work, i.e. the final compact form, which will be loaded by compute engines.

3.5 Graph Optimization: Removal of useless branches

In the computation on ciphertexts of a leveled HE scheme, we always find such a situation: mismatch of remaining levels in the two ciphertexts that will be added or multiplied together. In this case, we need to drop some extra moduli and their related RNSPolys of the ciphertext with higher level. In this case, some of the computation to generate the higher-level ciphertext would be wasted.

WorkBuilder in HLG framework has a basic graph optimization feature: removal of useless branches in the computing graph. Once we allow operator expansion to RNSPoly level, we can solve the wasted computation problem with little effort. As shown in Fig. 6, once we need to remove some modulus and related RNSPoly, all the extra computing graph generating the removed RNSPoly would be automatically removed. Thus we can save some computational cost automatically.

3.6 Automatic pre-computation in HLG framework

HLG framework supports automatic pre-computations. The constant operators/tasks set by user will be propagated to its successors and mark them as pre-computable. The pre-computation algorithm is as shown in Fig. 7.

This feature of pre-computation is very handy when we develop new operands and operators: it allows user to manage the pre-computation tables with little effort. For example, NTT operator requires pre-computed table to store the twiddle factors. For each of the different prime modulus, we need to generate different tables. The following code shows how we can use both features of pre-computation and operator deduplication to achieve easy management of pre-computed tables.

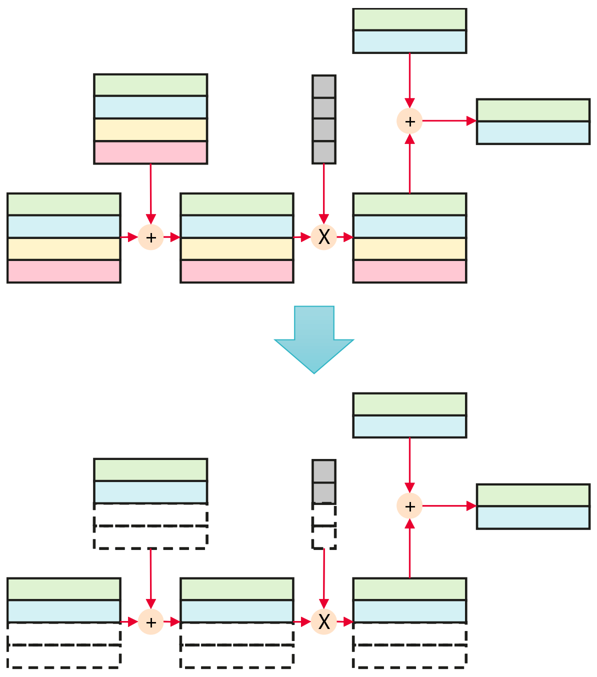


Figure 6: Automatic removal of useless branches in computing graph

```
RNSPolyNTT NttRns(RNSPoly a)
{
    RNSPolyNTT ret(a->Modulus());
    //The table below will be deduplicated and pre-computed automatically
    Vector table = ModifiedNttLutPreComputation(a->Modulus());

    MakeOperator(a->ContextPtr(), RlweOPType::NTT_LAZY_SCOTT_RNS_POLY,
                 ret, a->Modulus(), table, a);
    return ret;
}
```

In the example above, we call operator ModifiedNttLutPreComputation() on the modulus from input RNSPoly a and generated the output table. Then the pre-computation table will be used as one of the input when we construct the NTT operator. Then we can find that:

- There will always be one table only for each modulus, even if the NttRns function has been called many times. This is because of operator deduplication we has introduced in section 3.3.2.

- The data of the pre-computation table for each modulus will be automatically pre-computed and stored as part of the constants in the generated work. The pre-computation is triggered by the fact that the NTT moduli are already set as output of CONSTANT operators in CKKSContext by default.

As you can see from this example, the user of HLG doesn't need to care 1) how and when to pre-compute these table 2) where to store them and 3) how to find the correct table from the given modulus. We can also apply this trick when we implement Montgomery reduction [26], Shoup scalar multiplication [31] and other operators as long as they need pre-computation tables.

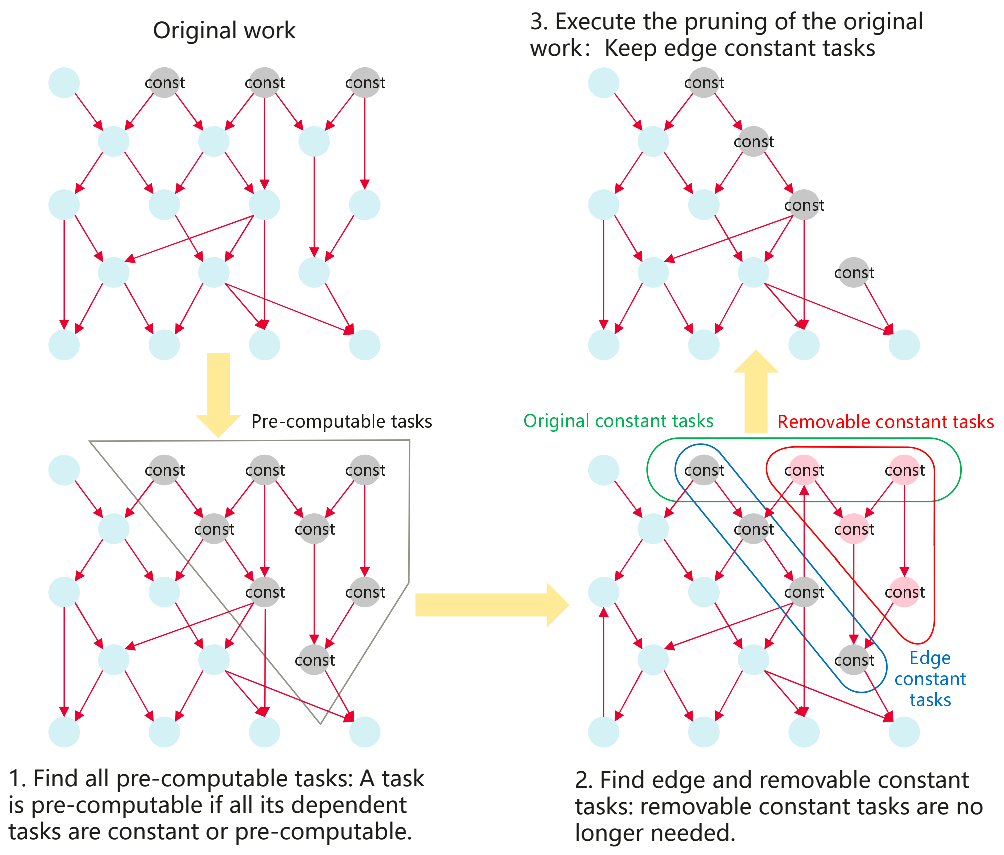

Original work
3. Execute the pruning of the original work: Keep edge constant tasks
const
Pre-computable tasks
Original constant tasks
Removable constant tasks
Edge constant tasks
1. Find all pre-computable tasks: A task is pre-computable if all its dependent tasks are constant or pre-computable.
2. Find edge and removable constant tasks: removable constant tasks are no longer needed.

Figure 7: Automatic pre-computation by propagation of constant tasks

3.7 Parameters and Context

The Parameters and the Context in HLG framework can both store information related to the computing graph under construction. However, there functionalities are different. The Parameters stores static settings which are universal and not related to specific computing graph. While the Context provides basic functionalities for the specific computing graph in construction. For example:

- **Parameters**: We can store RNS modulus in parameters. Since the same value of the modulus can be used to construct different computing graphs, the value of RNS modulus is considered to be "static".

  Note: here we can provide more examples of information which are considered "static" and can be stored in Parameters: polynomial degree, default relinearization key size, rotation index root, fft roots, level ranges for boosted key switch and pre-computation table for bit reversal, etc.

- **Context**: The Context contains a global map to store unique operand objects, which use the hash value of an operand object as the key. Then we can implement absorb operation and operator deduplication based on this global map. Since this map of unique operand objects is dependent on the construction process of the specific computing graph, it is not "static" and we should store it in the Context instead of the Parameters.

  Note: more examples of "non-static" information which are suitable to be stored in the Context: constant pool(mapping operand object to constant value), default scale factor, default RNS base, default configurations for operand types such as Ciphertext, Plaintext and RelinKey, etc.

We will explain functionalities of the Context in the following sections below.

3.7.1 Allocation of new placeholder

Context::NewPlaceholder() in the basic and unified interface to allocate placeholders, which is not supposed to be called by the user directly.

```
//size is the number of units in the placeholder
PlaceholderPtr NewPlaceholder(uint64_t size,
                              const std::string &name = "",
                              const std::string &description = "");
```

Besides that, HLG also provides another template function New<T> as a higher level interface to allocate placeholders.

```
template<class OperandType>
inline PlaceholderPtr New(ContextPtrType context,
                          const OperandConfig &config,
                          const std::string &name = "",
                          const std::string &description = "")
{
    ASSERT_BASE_TYPE(Operand, OperandType);
    CHECK_NULLPTR(context);
    auto size = Size<OperandType>(context, config);
    auto ret = context->NewPlaceholder(size, name, description);
    return ret;
}
```

The function New<T> will call Context::NewPlaceholder() interface, where the size of the placeholder is calculated from another template function Size<T>(). When implementing new operand types in HLG, we need to proved specialized template function of Size<T> (such as Size<ckks::Ciphertext>), and Config type (such as ckks::CiphertextConfig) too.

3.7.2 Creating new operators and allocate operator IDs

Context::NewOperator() is the basic and unified interface to create new operator in HLG, which is also not supposed to be called by user directly.

```
inline OperatorPtr NewOperator(int type,
                               const std::vector<PtrSyncerBasePtr> &inputs);
```

where the inputs are the PtrSyncer of the input operands for this operator. The Context will update the allocated operator ID each time this NewPperator() interface is called.

HLG also provide another variadic template interface MakeOperator, which is already introduced in section 3.3.2. Here we introduce more details of the interface itself.

```
template <class T, class... Args>
void MakeOperator(const ContextWeakPtrType &context,
                  int op_type,
                  OperandWrapper<T> &output,
                  const OperandWrapper<Args> &...inputs);
```

Note 1: This interface accepts one output and variant number of inputs (zero input is also acceptable). As mentioned in section 3.3.2, MakeOperator will trigger operator deduplication.

Note 2: The type of output and inputs are OperandWrapper<T>. As mentioned in section 3.2.7, all types provided to the user are derived from OperandWrapper<T>. You can put SingleUnit, RNSBase, RNSPoly, CRTPoly, ckks::Ciphertext as the output or the inputs. But after this operator is transformed into a Task by the WorkBuilder (introduced in section 3.4), all operand types will disappear and the output and inputs will become raw placeholders.

3.7.3 Providing default configurations for constructing operand types

Another useful functionality of the Context is to provided default configurations, so that we can write code in a more clean style. For example, after setting a Context as the Operand::DefaultContext() using interface Operand::SetDefaultContext(), we can construct ckks::Ciphertext, ckks::Plaintext and ckks::RelinKey without providing configurations.

```
/**
 * @brief Define a wrapper of class CiphertextImpl on the stack
 */
class Ciphertext : public OperandWrapper<CiphertextImpl> {
public:
    DEFINE_WRAPPER_METHODS(Ciphertext)

    explicit Ciphertext(
        const CiphertextConfig &config =
        Operand::DefaultContext<CKKSContext>()->DefaultCiphertextConfig(),
        const std::string &name = "",
        const std::string &description = "")
    {
        // default Ciphertext size = 2
        NewImplObj(config.rnsBase->ContextPtr(),
                   config, name, description);
    }

    ...
}
...
//usage of ckks:Ciphertext
//using default CiphertextConfig from default context
ckks::Ciphertext c;
```

Note that in this example, the method named as NewImplObj() is introduced in the next section.

3.7.4 Creating and registering all operand objects

The Context also provide functionality to create and register operand objects in a map of all unique operand objects, which will be used for operator deduplication.

```
//Update context's unique operand map according to operand object's hash
    value
//Side effect: if entry already exist, PtrSyncer's OperandPtr will be set to
     the existed one, for deduplication
bool RegisterUniqueOperand(OperandPtr &ptr);

//Template to create and register Operand objects, which will call
    RegisterUniqueOperand() for deduplication
//Returns ptr to operand object
template<class OperandImplType, class... Args>
inline auto NewOperandAndRegister(Args... args);

//Usage
ckks::CiphertextConfig config = Operand::DefaultContext<CKKSContext>()->
    DefaultCiphertextConfig();
auto ptr = context->NewOperandAndRegister<ckks::CiphertextImpl>(config);
```

When defining OperandWrapper types, user can use the macro defined in hlg_def.h: DEFINE_WRAPPER_METHODS(XXX). In this macro, the method NewImplObject is defined as:

```
template <class... Args>
void NewImplObj(const ContextWeakPtrType &context, Args... args)
{
    CHECK_NULLPTR(context);
    //create shared_ptr from weak_ptr
    auto ctx = context.lock();
    //create and register operand object and set pointer for the wrapper
    SetPtr(ctx->template NewOperandAndRegister<ImplType>(args...));
}
```

The user can just use NewImplObj interface to implement customized operand types, without using basic interface of the Context directly.

3.7.5 Constant management

There are three ways to set some operand as constants:

- The first way is to use WorkBuilder<T>::SetConstants() interface. This interface is used after construction of the original computing graph. Before we build the work from the original graph, we can set some input operand as constants by providing the constant value. Note that setting constant operand will trigger auto pre-computation. We can find this usage in the example of ckks_hello_world.cpp

```
...
// ------- define high-level algebraic objects ---------
ckks::Plaintext PI, c1, c0;
ckks::Ciphertext x;
ckks::RelinKey rlk;
// ------------- define computation graph --------------
...
// ---------------- prepare constants ------------------
ckks::Encoder encoder(params);
// msg encode for constants
ckks::PlaintextData PI_data = encoder.EncodeDoubleConst(3.1415926);
ckks::PlaintextData c1_data = encoder.EncodeDoubleConst(0.4);
ckks::PlaintextData c0_data = encoder.EncodeDoubleConst(1.0);
// --------- build work from computing graph -----------
WorkBuilder<uint64_t> builder(context);
auto work = builder.SetInput(x, rlk)
                   .SetConstant(PI, PI_data)
                   .SetConstant(c1, c1_data)
                   .SetConstant(c0, c0_data)
                   .SetOutput(output, output->ScaleFactor())
                   .Build();
...
```

- The second way to set constant operand is to use Context::SetConstants() method. This method is used before WorkBuilder<T>::Build(). However, it will temporarily store the operand and the const value in a const pool and the WorkBuilder will call SetConstant() method for each pair in the constant pool automatically.

```
//set constant for single-unit sized operand
template<class T, class ValueType>
inline void SetConstants(const OperandWrapper<T> &oprd,
                         const ValueType &value);
//set constant for multi-unit sized operand
template<class T, class UnitType>
inline void SetConstants(const OperandWrapper<T> &oprd,
                         const std::vector<UnitType> &vec);
...
//Usage
ckks::Plaintext p;
context->SetConstants(p, encoder.Encode(2.0));
WorkBuilder<uint64_t> builder(context);
```

```
//no need to call SetConstant() for p again here
auto work = builder.SetInput(...)
                   ...
                   .Build();
```

- The third way to set constant operand is to use Context::GetConstantOperandFromValue() methods. This method will create new operand and then set them as constants by calling Context::SetConstants(), which is easier to use. Note: the ValueType here must be 64-bit now, such as uint64_t, int64_t and double etc. Otherwise there will be problems, since we only support 64-bit unit cells for the placeholder now.

```
//Get constant operand from single value
template<class OperandType, class ValueType>
OperandType GetConstantOperandFromValue(const ValueType &value);

//Get constant operand from value vector for multiple placeholder units
template<class OperandType, class ValueType>
OperandType GetConstantOperandFromValueVector(
                const std::vector<ValueType> &vec);

//Get constant operand from std::complex<T>-typed values
template<class OperandType, class ValueType>
OperandType GetConstantOperandFromValueVector(
                const std::vector<std::complex<ValueType>> &vec);

//Usage: type of BitReverseVector is std::<uint64_t>, each unit cell is
     64-bit long
auto para = context->Parameters();
auto BIT_REVERSE_VEC = context->GetConstantOperandFromValueVector<
    Vector>(para->BitReverseVector);
```

  Note that the new operand BIT_REVERSE_VEC created in this example will be deduplicated if it is created by calling GetConstantOperandFromValueVector<T>() again using the same Operand type Vector and the same constant value para->BitReverseVector;

There are also very useful methods to read constant value from constant operand:

```
//Get constant value from constant operand (placeholder size == 1)
template<class ValueType, class OperandType>
ValueType GetConstantValueFromOperand(const OperandType &oprd);

//Get constant vector from constant operand (placeholder size > 1)
template<class ValueType, class OperandType>
std::vector<ValueType> GetConstantVectorFromOperand(const OperandType &oprd)
    ;

//Get constant vector of std::complex<T> type from constant operand
template<class ValueType, class OperandType>
std::vector<std::complex<ValueType>> GetConstantComplexVectorFromOperand(
    const OperandType &oprd);
```

There is an example of reading value from constant operand to switch between different versions of implementations of INTT operator depending on the bit-length of the modulus value. For non-constant operand, we cannot obtain their value during the process of construction of the computing graph.

```
//Usage: example from implementation of INTT operator
RNSPoly InttRns(RNSPolyNTT a)
{
    RNSPoly ret(a->Modulus());
    Vector table = ModifiedNttLutPreComputation(a->Modulus());

    auto context = a->ContextPtr();
    auto polyDegree = static_cast<uint64_t>(context->Parameters()->
        polyDegree);
    //read value of modulus from constant operand a->Modulus()
    auto modVal = context->GetConstantValueFromOperand<uint64_t>(a->Modulus
        ());
    //use the value in the if statement
    if (LogTwo(modVal) + LogTwo(polyDegree) >= MODIFIED_INTT_THRESHOLD) {
        MakeOperator(a->ContextPtr(), RlweOPType::INTT_HYBRID_RNS_POLY_NTT,
            ret, a->Modulus(), table, a);
    } else {
        MakeOperator(a->ContextPtr(), RlweOPType::INTT_SCOTT_RNS_POLY_NTT,
            ret, a->Modulus(), table, a);
    }
    return ret;
}
```

3.8 Serialization and compression of data

3.8.1 Raw data for Plaintext, Ciphertext and Keys

Serialization is relatively simple in HLG framework. From the perspective of the framework itself, all data are regarded as raw data of UnitType, which is uint64_t or uint32_t, etc. We have defined types for these raw data: PlaindextData, CiphertextData, RelinKeyData, etc., which are all derived from std::vector<UnitType>. We only need to consider endians on different platforms when we serialized data of UnitType.

Only when these raw data are used as input in the implementation of primitive operators, these implementations will recognize specific types of the raw data, which is implemented using std::reinterpret_cast<T*>() of c++.

In HLG framework, we provide interfaces to serialize and deserialize raw data:

```
//serialization
void SerializeToOstream(const std::vector<uint64_t> &data,
                        std::ostream &os);
void SaveRawDataToFile(const std::vector<uint64_t> &data,
                       const std::string &filename);
//deserialization
std::vector<uint64_t> DeserializeRawDataFromIstream(std::istream &is);
std::vector<uint64_t> LoadRawDataFromFile(const std::string &filename);
```

3.8.2 Serialization of work

Serialization of work is implemented using the library of ProtoBuffer by google [32]. We defined data type ProtoWork with ProtoBuffer as the serializable alternative form of Work, we also provided interfaces to serialize work:

```
//serialization
void ToProto(ProtoWork &ret, const Work<uint64_t> &work);
void SerializeToOstream(const Work<uint64_t> &work, std::ostream &os);
void SaveWorkToFile(const Work<uint64_t> &work,
                    const std::string &filename);

//deserialization
ProtoWork DeserializeProtoWorkFromIstream(std::istream &is);
ProtoWork LoadProtoWorkFromFile(const std::string &filename);
```

Note 1: HLG does not support converting ProtoWork back to Work, since this seems unnecessary now.

Note 2: The CPU engines in HLG framework can support interfaces to load from a Work and a ProtoWork.

3.8.3 Compression of raw data

Almost all data in 2^{nd}-generation FHE are polynomials, whose coefficients are modular. We have to use $uint64\_t$ to store their value, even if the modular prime is only 40-bit or less. So compression is necessary to remove those zeros in the MSBs of the 64-bit values, which is also good for reducing the data size transferred through internet. The data compression and decompression in HLG is implemented using third-party library Zlib [17].

In serialization and deserialization of the work mentioned before, compression and decompression are also supported.

3.9 APIs for implementation of primitive operators

In HLG framework, implementation of primitive operators is also decoupled with the strategy for task scheduling and memory management in CPU compute engines. In order to achieve this, the CPU engines should provide unified interface DataAccessAPI to the implementation code for primitive operators, from which the underlying code can obtain address of the memory allocated for the input and output placeholders of the operator.

The memory access obtained from DataAccessAPI will be a custom type VirtualSpan<T>. In order to define VirtualSpan<T>, we need to define SizedPointer<T> first.

3.9.1 SizedPointer<T>: secure pointer type

SizedPointer<T> is a secure pointer type, with members of the raw pointer of type T* and a data size. Usage of a SizedPointer<T> p is similar to raw pointer: *p, p[2], p++, etc. However there is additional check for range boundary. Basically, SizedPointer<T> can be regarded as a lightweight data view of a **continuous** range of memory space, which is similar to std::span in C++20. Copy of SizedPointer<T> will always be shallow copy.

The examples of usage of SizedPointer<T> is as follows:

```
//original data vectors and arrays
std::vector<uint64_t> a{0,1,2,3};
int b[4] = {4,5,6,7};
std::array<uint64_t> c = {8,9,10,11};

//construction of SizedPointer from vector
SizedPointer<uint64_t> sp1(a);
//construction of SizedPointer from raw pointer and size
SizedPointer<uint64_t> sp2(b, 4);
//construction of SizedPointer from std::array
SizedPointer<uint64_t> sp3(c);

//get raw pointer
assert(sp1.Ptr()==a.data());
assert(sp2.Ptr()==b);
assert(sp3.Ptr()==c.data());

//access by operator[], complexity O(1)
assert(sp1[1]==1);
assert(sp2[1]==5);
assert(sp3[1]==9);

//access as a pointer
*sp1 = 2;                //write to sp1[0]
assert(sp1[0]==2);
assert(*(sp1+2)==2);     //*(sp1+2)=sp1[2]
sp1+=3;
assert(*sp1==3);

//will trigger error: out of range
auto v = sp1[4];
```

3.9.2 VirtualSpan<T>: a virtual continuous memory range

Since HLG supports customizable granularity of operator expansion, sometimes the input and/or the output of an operator can be either a whole piece, or fragmented pieces. In order to make the life of developer easier, we need a unified interface to support both cases with the same interface for memory access.

In order to support this, we introduce another type VirtualSpan<T>, which can be constructed using a whole piece of memory or several memory pieces. However, the interface to access the data in a VirtualSpan is unified:

```
//definitions of vectors
std::vector<uint64_t> a{0,1,2,3};
std::vector<uint64_t> b{4,5,6,7};
std::vector<uint64_t> c{8,9,10,11};

//construct sized pointer from vector c
SizedPointer<uint64_t> spa(a);
SizedPointer<uint64_t> spb(b);
SizedPointer<uint64_t> spc(c);

//definition of VirtualSpan from multiple vectors and sized pointers
VirtualSpan<uint64_t> vs(a,b,spc);

//access by operator[], complexity O(log(n))
assert(vs[2]==2);   //vs[2]==a[2]
assert(vs[5]==5);   //vs[5]==b[2]
assert(vs[9]==9);   //vs[9]==sp[2]==c[2]

//access by iterator
auto it = vs.begin();
uint64_t i = 0;
for(;it!=vs.end();it++){
    //complexity of it++ is O(1)
    assert(*it==i++);
}

//access by range for loop
i = 0;
for(auto &v:vs){           //implicit: complexity of it++ is O(1)
    assert(v==i++);
}

//access by zip range of multiple VirtualSpans
VirtualSpan<uint64_t> vs2(spa,spb,c);   //another virtual span
//loop by Zip range will terminate when any one of the range reaches the end
for(auto &&[v1, v2]:Zip(vs1, vs2)){
    assert(v1==v2);
}
```

In addition, we also support custom MemCopy function between SizedPointers and between VirtualSpans. However, VirtualSpan will have worse performance comparing to SizedPointer. We will only consider using VirtualSpan in non-performance-critical operators, such as operators which will always be used in pre-computation phase, such as InvModRnsBase2Crt and other operators computing constants from modulus values.

3.9.3 DataAccessAPIBase<T>: unified interface for implementation of primitive operators

DataAccessAPIBase<T> is the base class which provides interface to obtain input and output data as SizedPointer<T> or VirtualSpan<T>. The following interfaces are provided:

- GetOutput<T>(): get output SizedPointer<T>, which is writable.

- GetInput<T, U, V, ...>(): get multiple SizedPointers as a tuple of types const T, const U, const V, ... which are read-only.
- GetOutputV<T>(): get output VirtualSpan<T>, which is writable.
- GetInputV<T, U, V, ...>(): get multiple VirtualSpans as a tuple of types const T, const U, const V, ... which are read-only.

Here we use implementation of primitive operator AddModRnsPoly as an example:

```
void OpAddModRnsPoly(const DataAccessAPIBase& api) {
    //ret is a SizedPointer<uint64_t>, where we can write to output data
    auto ret = api.GetOutput<uint64_t>(); // --- mont_form

    //modulus, poly1 and poly2 are SizedPointer<const uint64_t> (read-only)
    auto[modulus, poly1, poly2]
                    = api.GetInput<uint64_t, uint64_t, uint64_t>();

    assert(poly1.Size() == poly2.Size());
    assert(poly1.Size() == ret.Size());

    //use SizedPointer as raw pointer / original array in C++
    for (size_t i = 0; i < poly1.Size(); i++) {
        ret[i] = FastAddMod(poly1[i], poly2[i], *modulus);
    }
}
```

Note that in existing CPU engines of HLG, all implementation of operators are required to support the same interface as $OpAddModRnsPoly$ above.

3.10 Compute engines for multiple CPU cores

The compute engines of HLG are designed to be generic. They need to support all kinds of computing graphs, i.e. works, with automatic task scheduling and memory management.

In the data structure of the Work, we have defined dependency set and notification set of a task t, which are useful for task scheduling and memory management.

In HLG framework, we provide two different implementations of CPU engines: MyEngine and LockFreeEngine, which follow similar strategies for task scheduling. However MyEngine is using mutex when updating dependency and notification counters and allocating memory, etc, while LockFreeEngine is implemented based on lock-free queues. The end-to-end performance of MyEngine and LockFreeEngine are similar. Their performance may vary for different kind of works. We will need experiments to find out which engine is better for the given work.

The usage of both engines are the same. The interfaces are as follows:

```
//step 0: Initialization. Create context for computing graph.
...
//step 1: Construct high-level computing graph
ckks::Ciphertext a,b;
ckks::Ciphertext c = a*b | ckks::Relin | ckks::Rescale;

//step 2: Build work using WorkBuilder
//context contains the entire high-level computing graph
WorkBuilder<uint64_t> builder(context);
//build work
auto work = builder.SetInput(a,b)
                   .SetOutput(c)
                   .Build();

//step 3: Execute the work using given cpu engine
//define MyEngine with 8 threads
MyEngine<uint64_t> engine(8);   //or LockFreeEngine<uint64_t> engine(8);
//load work into engine
```

```
engine.load(work);
//run engine with real input data
//data_a and data_b: real input data for operand a and b
auto output = engine.Run(data_a, data_b);
//type of output is std::vector<std::vector<uint64_t>>
//data_c is the real output data for operand c
ckks::CiphertextData data_c = output[0];
```

The user of HLG may also implement their own engines with different strategies for task scheduling and memory management, since the architecture of compute engine is also decoupled from the front-end algorithm and implementation of primitive operators.

3.10.1 Description of MyEngine

The dependency set of the task t stores the IDs of all the tasks which task t depends on, which means those tasks need to be executed before task t. The strategy of task scheduling and memory management is as follows:

1. Find all tasks with empty dependency set (the executable tasks) and push these tasks into the queue of the thread pool.

2. For each of the worker thread in the thread pool, take one task from the task queue, allocate memory for their output placeholders and execute the task by calling the implementation code of this task.

3. After execution of the task t is finished, we can notify the tasks in the notification set of t. Then these successor tasks of t can update their dependency set by removing t from it. If the dependency set of any successor task become empty, we also push the successor task into the thread pool.

4. Also, after execution of task t, we can ask the predecessor tasks in the dependency set of task t to remove t from their notification set. Once the notification set of any predecessor task becomes empty, it means that the output data of this task is no longer needed. Thus we can deallocate the memory for their output placeholders.

   Note that we have to exclude task types INPUT, CONSTANT operators and also the output tasks of the work when deallocating memory.

In MyEngine, all threads in the thread pool are equivalent. They will perform computation of the assigned task and memory management, updating of counters and pushing tasks into thread pool alternatively. In short, it is a multi-producer and multi-consumer design. When the threads in the thread pool are working, the main thread would be idle, waiting for them to finish. The architecture of MyEngine is illustrated in Fig. 8.

3.10.2 Description of LockFreeEngine

For the other CPU engine LockFreeEngine in HLG framework, these strategies of task scheduling are similar. However there is a main thread in LockFreeEngine in charge of three operations: 1) allocating memory 2) updating reference and notification counters and push tasks into runnable task queue 3) releasing memory. The architecture of LockFreeEngine is introduced in Fig. 9.

4 Implementation of CKKS based on HLG framework

Base on HLG framework, we have implemented operand types for RLWE and higher-level types for CKKS. The hierarchy among operand types are illustrated in Fig. 10. Note that parent types are at higher position.

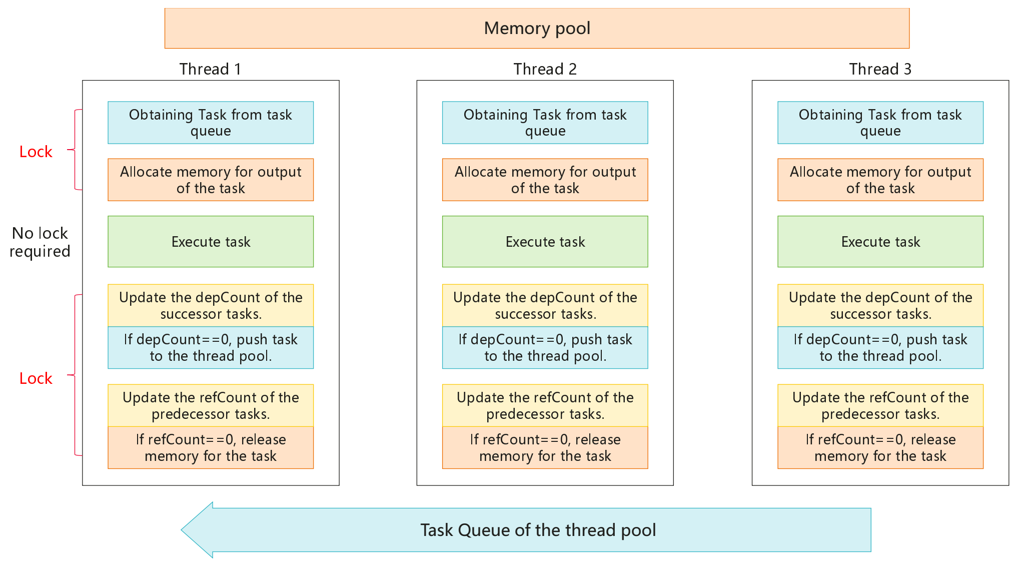

Memory pool
Thread 1
Thread 2
Thread 3
Lock
No lock required
Lock
Obtaining Task from task queue
Allocate memory for output of the task
Execute task
Update the depCount of the successor tasks.
If depCount==0, push task to the thread pool.
Update the refCount of the predecessor tasks.
If refCount==0, release memory for the task
Task Queue of the thread pool

Figure 8: Architecture of MyEngine

As is shown in the bottom of Fig. 10, primitive operand types are provided in HLG framework, including SingleUnit, Double, Complex and Vector type. As their name show, the SingleUnit refers to an operand with raw data size being 1, and Vector type is defined as an array of SingleUnit. Double is the data type in HLG corresponding to the double value, for example, the scale factor in CKKS is of type Double in our framework. Complex type is an array of Double, with size being 2.

Based on the basic types, we defined new data types and operators to support for RLWE related algorithms. Unlike these basic operand types, the data types in RLWE and CKKS require more complex information. For example, the design of CRTPolyNTT needs the RNS base information, and the ckks::Ciphertext that inherits from CRTPolyNTT requires the RNS base, the scale factor and the size of a ciphertext. Therefore we define the corresponding xxxConfig struct to describe the required configuration, and it helps in defining a new type and calculating the size of placeholder. The diagrams of xxxConfig is similar with the operand type dependencies.

For the construction of an operand, there are the following methods that are applicable to different scenarios.

- **Construct from its config.** This construction mode is applicable to the scenario where a new operand needs to be created. For example, when defining the input operand of a task, we usually use this construction method.
- **Construct from children.** It is used to create a new operand, i.e. the "step parent" from the collection of existing children. Note its placeholder must be virtual.
- **Construct from existing placeholder and children's Config.** This constructor is called during the execution of MakeChildren(), that is to cut a small piece from the placeholder that has been allocated by the birth parent and assign it to the child operand. Note that besides the placeholder input, the additional Config of the child operand is also required.
- **Construct from base class object.** For example, we can use a CRTPolyNTT object to construct a CRTPoly.

Based on the defined operands, we define the related operators that are illustrated in Fig.11. Note that the operators in red are primitive and implemented. As is shown in the

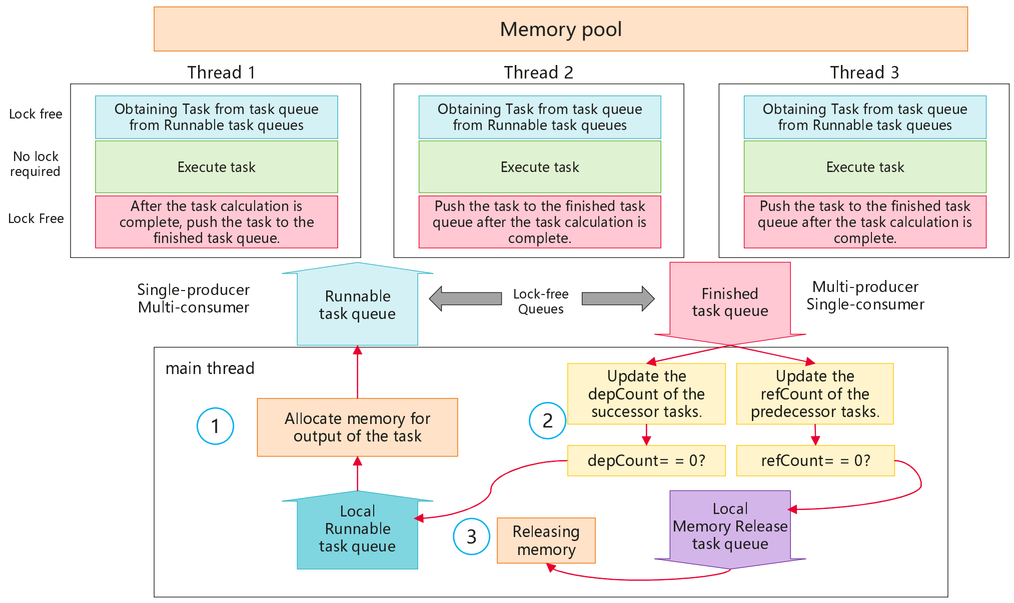

Memory pool
Thread 1
Thread 2
Thread 3
Lock free
No lock required
Lock Free
Obtaining Task from task queue from Runnable task queues
Execute task
After the task calculation is complete, push the task to the finished task queue.
Push the task to the finished task queue after the task calculation is complete.
Single-producer Multi-consumer
Runnable task queue
Lock-free Queues
Finished task queue
Multi-producer Single-consumer
main thread
1
Allocate memory for output of the task
2
Update the depCount of the successor tasks.
Update the refCount of the predecessor tasks.
depCount= = 0?
refCount= = 0?
Local Runnable task queue
3
Releasing memory
Local Memory Release task queue

Figure 9: Architecture of LockFreeEngine

bottom of Fig.11, we provide the pre-defined basic operators to support the generation of random value, add/substraction of Double and Vector type. On top of this, we extend the operators of RLWE and CKKS. We give the description of main operators in the following subsections and the reader can refer to the code for comprehensive introduction of all operators.

4.1 Operand Types and Operator Types for RLWE

4.1.1 Operand Types for RLWE

- **RNSBase and CRTConst.** The RNSBase type is defined as an array of SingleUnit that has a placeholder to store the RNS modulus. It can be constructed either from the RNSBaseConfig, i,e, the raw data size information or from its children, i.e. vector of SingleUnit. For example, we can construct a RNSBase of size 4 as follows.

```
RNSBase base0(4);   //construct from raw data size

SingleUnit a,b,c,d;
RNSBase base1({a,b,c,d});  //construct from children
```

  The methods provided by RNSBase type includes the combination and extraction related functions. For example,

```
//get index-th Modulus of an RNSBase object
SingleUnit GetModulus(uint32_t index);

//construct a new RNSBase with the remaining modulus
RNSBase RemoveRnsBase(const RNSBase &base);

//find intersection (as RNSBase-type) of this RNSBase with target base
RNSBase Intersection(const RNSBase &base);

//append modulus in base to current modulus set
RNSBase Append(const RNSBase &base);

RNSBase FirstN(uint32_t n); // Get first n modulus as a new RNSBase
```

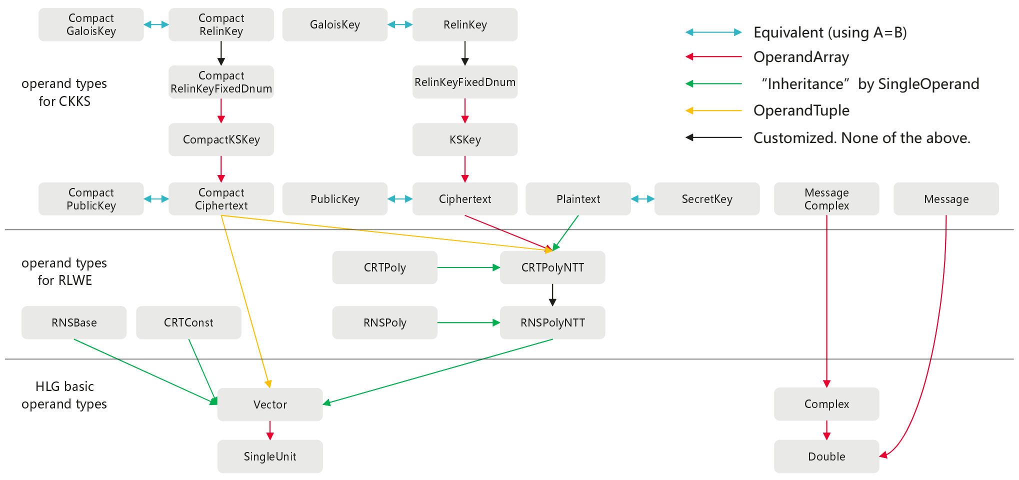


Figure 10: Hierarchy of operand types for CKKS

For more comprehensive information, the reader can refer to the code.

Similar with the RNSBase type, the CRTConst type is defined as an array of SingleUnit. The CRTConst type is defined to store the CRT representation of a constant which is related with the RNS base. We list an example of the construction of a CRTConst here.

```
RNSBase base(4);
CRTConst r(base);   //construct from RNS base
```

The operator [ ] is also given to get the i-th child, i.e. a SingleUnit of modulus.

- **RNSPolyNTT and CRTPolyNTT.** The RNSPolyNTT type is inherited from the SingleOperand template of Vector type and defined to represent the NTT form of a RNS polynomial. Compared to a single Vector, it requires an additional modulus to construct and the vector size is fixed which is equal to the polynomial degree.

  The definition of CRTPolyNTT is to store the NTT form of a CRT polynomial and its children is RNSPolyNTTs. Note that we do not use OperandArray<RNSPolyNTT> to define the type, since the template OperandArray requires the configuration of each child to be the same. But for CRTPolyNTT, each child is of different modulus, and we define CRTPolyNTT from Operand directly.

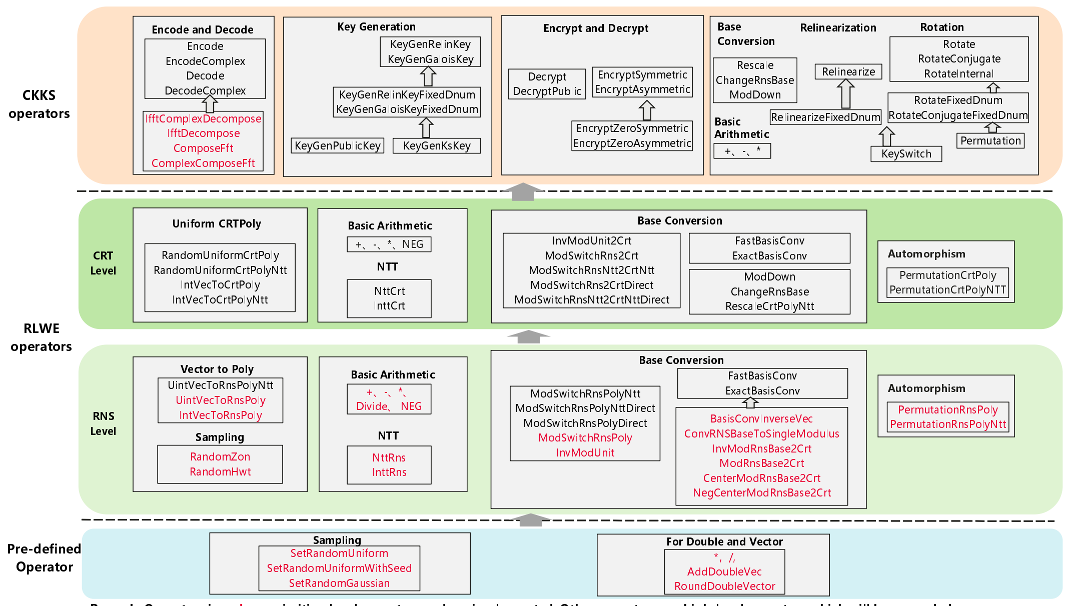
  We also give the methods of CRTPolyNTT to support splitting and combination friendly, for example,

```
//remove given RNSBase related RNSPolyNTT's
CRTPolyNTT RemoveRnsBase(RNSBase base);

//remove last n related RNSPolyNTT's and return a new CRTPolyNTT object
CRTPolyNTT RemoveLastNMod(uint32_t n);

RNSPolyNTT GetSubRnsPolyNttByModulus(SingleUnit modulus);
```

- **RNSPoly and CRTPoly.** Since the NTT form and non-NTT form of a polynomial are all involved in FHE algorithm, we also define the RNSPoly and CRTPoly type that are inherited from RNSPolyNTT and CRTPolyNTT respectively. Similar methods of splitting and combination are provided.



Remark: Operators in red are primitive-level operators and are implemented. Other operators are high-level operators which will be expanded.

Figure 11: Operator Hierarchy Diagram

4.1.2 Operator Types for RLWE

We defined the commonly used operators for the operands. Note the operator of RNSPoly/RNSPolyNTT is primitive-level and implemented. Based on that, the high-level operators, such as operators of CRTPoly, are further given.

- **Modular addition/negation/multiplication.** We define the basic operators for each operand and provide the convenient representation by using operator $+, -, *$, that is for operand T,R $\in \{RNSPoly, RNSPolyNTT, CRTPoly, CRTPolyNTT\}$, we construct the following operators.

```
T operator+(T a, R b);
T operator-(T a, R b);
T operator-(T a);
T operator*(T a, R b);
```

  Take the multiplication of CRTPolyNTT as an example, the user can write as follows.

```
CRTPolyNTT a;
CRTPolyNTT b;
CRTpolyNTT ret = a * b;
```

  Note that when we deal with two operands, they should have same context and matching modulus.

- **NTT/INTT.** As for the transformation between NTT form and non-NTT form, we provide the NTT/INTT operator for RNSPoly and CRTPoly.

```
RNSPolyNTT NttRns(RNSPoly a);
RNSPoly InttRns(RNSPolyNTT a);
CRTPolyNTT NttCrt(CRTPoly a);
CRTPoly InttCrt(CRTPolyNTT a);
```

- **Mod Switch.** We implement the following two operators of RNSPoly.

The ModSwitchRNSPoly operator is used for mod switching for RNSPoly with tuning of symmetric range, i.e. we consider the coefficient in the range of $(-(q-1)/2, (q-1)/2]$ instead of $[0, q)$ where q is a modulus.

As a contrast, the ModSwitchRNSPolyDirect operator considers the coefficient in the range of $[0, q)$ and its expand function is defined by setting a new mod.

```
RNSPoly ModSwitchRNSPolyDirect(RNSPoly a, SingleUnit mod);
RNSPoly ModSwitchRNSPoly(RNSPoly a, SingleUnit mod);
```

The user can choose the corresponding switch operator due to the noise control requirement.

Based on the two operators, the high-level mod switch operators are also given. We list the ModSwitch operators here. And the ModSwitchDirect's high level operators are similar.

```
//mod switch to RNSPolyNTT, from RNSPolyNTT
RNSPolyNTT ModSwitchRNSPolyNtt(RNSPolyNTT a, SingleUnit mod);

//mod switch to CRTPoly, for CRTPoly
CRTPoly ModSwitchRns2Crt(RNSPoly a, RNSBase base);

//mod switch to CRTPolyNTT, for RNSPolyNTT
CRTPolyNTT ModSwitchRnsNtt2CrtNtt(RNSPolyNTT a, RNSBase base);
```

- **Rescale.** The rescale operator for CRTPolyNTT is shown in Fig.4 and it is assembled by defined operators such as modular addition, NTT/INTT and mod switch etc.

```
CRTPolyNTT RescaleCrtPolyNtt(CRTPolyNTT a);
```

- **Basis Conversion.** We define the RNS base conversion related operators to switch the poly from its original RNSBase to a new RNSBase.

  - **Fast Basis Conversion.** The FastBasisConv operator is defined to converse a CRTPoly into a RNSPoly with a new modulus. Recall the fast basis conversion algorithm [5], it firstly performs a scalar multiplication for each CRT component, then calculate the scalar multiplications and additions under the new modulus. Our assembly function of FastBasisConv is consistent with that and is shown in Fig.12.

```
RNSPoly FastBasisConv(CRTPoly a, SingleUnit mod);
```

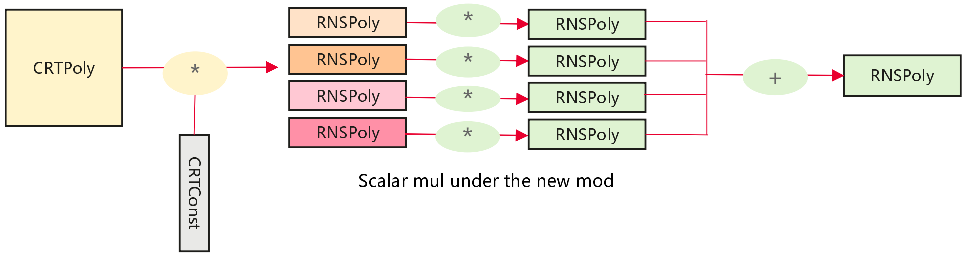
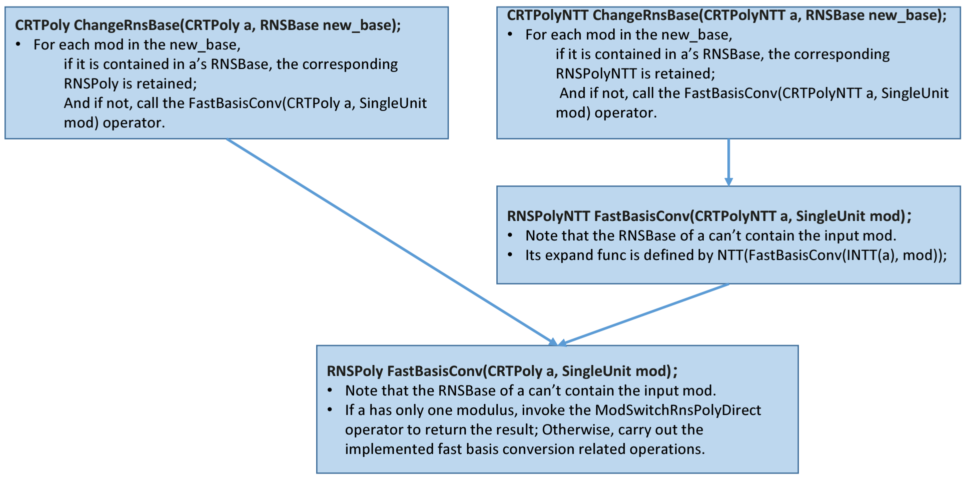


Figure 12: The detail chart of FastBasisConv operator

  - **Exact Basis Conversion.** Note that the fast basis conversion would introduce approximation error and the exact basis conversion may needed in some cases. We provide the ExactBasisConv operator to meet the requirements. We implement the following primitive operator and assemble it into high-level operators similarly.

```
RNSPoly ExactBasisConv(CRTPoly a, SingleUnit mod);
```

– **RNS Base Change.** We define the ChangeRnsBase related operators to switch the poly from its original RNSBase to a new RNSBase. Regardless of whether the new base and the original base have intersections, ChangRnsBase operator is supported and is shown in Fig.13.



CRTPoly ChangeRnsBase(CRTPoly a, RNSBase new_base);
• For each mod in the new_base,
if it is contained in a's RNSBase, the corresponding RNSPoly is retained;
And if not, call the FastBasisConv(CRTPoly a, SingleUnit mod) operator.
CRTPolyNTT ChangeRnsBase(CRTPolyNTT a, RNSBase new_base);
• For each mod in the new_base,
if it is contained in a's RNSBase, the corresponding RNSPolyNTT is retained;
And if not, call the FastBasisConv(CRTPolyNTT a, SingleUnit mod) operator.
RNSPolyNTT FastBasisConv(CRTPolyNTT a, SingleUnit mod);
• Note that the RNSBase of a can't contain the input mod.
• Its expand func is defined by NTT(FastBasisConv(INTT(a), mod));
RNSPoly FastBasisConv(CRTPoly a, SingleUnit mod);
• Note that the RNSBase of a can't contain the input mod.
• If a has only one modulus, invoke the ModSwitchRnsPolyDirect operator to return the result; Otherwise, carry out the implemented fast basis conversion related operations.

Figure 13: ChangeRnsBase operators

We take an example to interpret the definition and implementation as is shown in Fig. 14. For a CRTPoly a with its RNSbase being p_0, p_1, p_2, p_3 and it needs to be changed into new RNSBase p_0, q_1, q_2, the returned CRTPoly would retain the RNSPoly $a[0]$ and generate the RNSPoly of q_1, q_2 by carrying on FastBasisConv operators.

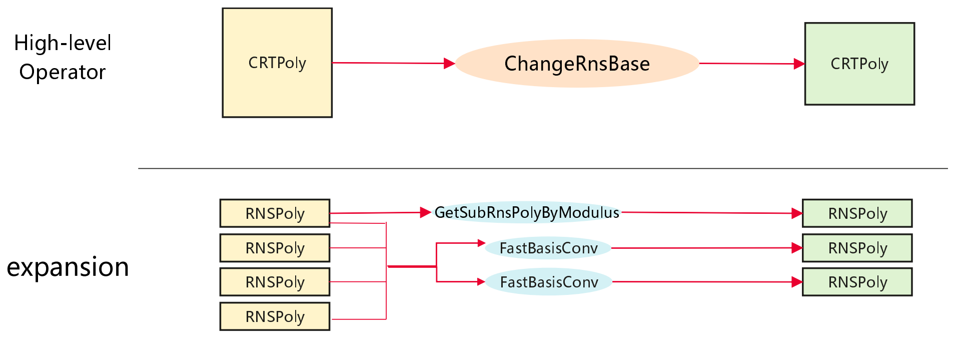


Figure 14: The expansion of ChangeRnsBase operator

• **Mod Down.** For a CRT poly a with modulus being PQ, the mod down function is to calculate $\frac{a}{P}$ which can be seen as a general Rescale, since the divided modulus can be composed of several RNS modulus, instead of a single modulus. We provide the following interfaces. And its expanding process is shown in Fig. 15. Note that we can select FastBasisConv or ExactBasisConv to perform intermediate basis conversion due to the requirements of noise control.

```
//the move_base should be included in a's RNS base.
CRTPoly ModDown(CRTPoly a, RNSBase move_base);
CRTPolyNTT ModDown(CRTPolyNTT a, RNSBase move_base);
```

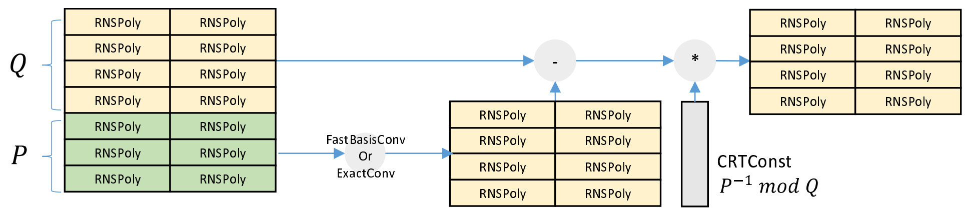


Figure 15: The expansion of ModDown operator

4.2 Operand Types and Operator Types for CKKS

4.2.1 Operand Types of CKKS

Based on the operand types of RLWE, we construct the CKKS algorithm related types as shown in Fig. 10. In addition to the operands specified, all other operands class support three main types of constructors, that is construct from Config, construct from its children, and construct from the existing placeholder and its child config.

- **Message and MessageComplex.** The Message is the operand that corresponds to the double message and it is defined as an array of Double. Similarly, the MessageComplex is defined as array of Complex and is for the complex messages. The user can choose the corresponding operand according to the type of message.

  The MessageConfig/MessageComplexConfig is inherited from VectorConfig and the size of message is needed. Note that in the construction of Message/MessageComplex, we only need the constructor from the config. And the default Message/MessageComplex config is the slot size.

- **Plaintext.** The plaintext of CKKS algorithm is in the NTT form by default, and the Plaintext class is defined by inheriting from the SingleOperand template of CRTPolyNTT. Besides the RNS base, the scale factor is also required in PlaintextConfig.

  Note that the default constructor of Plaintext is to use the default PlaintextConfig that provided by DefaultPlaintextConfig() method in CKKSContext, i.e. we can write as follows to construct a Plaintext.

  ```
  //here it is with the default RNSBase and the default Scale factor.
  ckks::Plaintext p;
  ```

  Note that in the definition of the CKKS operands,such as Plaintext,Ciphertext,KSKey etc, the commonly used methods of splitting and combination, for example removing a part or matching the RNS base, are all provided. Note that we also provide the method for setting scale factor.

  ```
  //Set the scale factor of a Plaintext
  std::function<ckks::Plaintext(ckks::Plaintext)> SetScaleP(Double
      scale_factor);
  ```

  For more information, the reader can refer to the code.

- **SecretKey.** The SecretKey operand is equivalent with Plaintext that this alias is used to represent the secret key conveniently.

- **Ciphertext.** The Ciphertext operand is defined by inheriting from OperandArray template of CRTPolyNTT. Since besides the RNS base, the ciphertext size and the scale factor are also required in describing a ciphertext, they are all included in the CiphertextConfig and the member of Ciphertext class. If we want to construct a default Ciphertext, we can write as follows.

```
//here it is with the default RNSBase, the default Scale factor and the
     default ciphertext size.
ckks::Ciphertext c;
```

Remark that the RandomSeed of type Vector is also a member of Ciphertext class, and it is only used for fresh symmetric encrypted ciphertexts, i.e. we can use $RandomSeed()$ to get the seed of the random part of a Ciphertext.

- **PublicKey.** The PublicKey is equivalent to Ciphertext operand as it is an fresh ciphertext of encrypting zero.

- **KSKey.** The KSKey operand is defined for the switching keys. Recall that the switching key is composed of several ciphertexts, the KSKey class is defined as an array of Ciphertext operand.

  To support the boosted key switching and automatically select the corresponding key to switch, the following information is included in describing the KSKey configuration.

```
/**
 *@param extra_key_rns_base_, the extra base in the switching key;
 *@param ciphertext_rns_base_, the normal ciphertext rns base;
 *@param ciphertext_size_ ,
         the size of CRTPolyNTTs in each child Ciphertext of a KSKey
 *@param scale_factor_, used for the ciphertext setting
 *@param dnum,  represents the digit number of the KSKey, the digit can
      range from 1 to L+1.
 *@param max_level, shows the max level that the KSKey support.
**/
KSKeyConfig(RNSBase extra_key_rns_base_,
            RNSBase ciphertext_rns_base_,
            uint32_t ciphertext_size_,
            const Double scale_factor_,
            const uint32_t dnum,
            const uint32_t max_level)
```

  Note that the digit number and the support switching max level are used for the carry on the key switching correctly. We give more interpretation of the max_level parameter. For example, for ciphertexts of total level being 10, if the max_level of a KSKey is 6, then it can only support the ciphertext of level $[0, 6]$ to carry on the key switching.

- **RelinKeyFixedDnum and RelinKey.** The RelinKeyFixedDnum operand is defined for the relinearization key of a fixed digit number. Based on this, the RelinKey type is defined which is composed of several RelinKeyFixedDnum. The type is defined to support automatically match the corresponding key.

  For example, for the CKKS parameters with total level being 10, the user may need the relinearization key of digit 2 for ciphertexts with level ranging from 0 to 6, and the relinear key of digit 1 for ciphertexts with level ranging from 7 to 10. Then we can define two RelinKeyFixedDnum operands with digit being 2 and 1 respectively, and the two RelinKeyFixedDnum operands form a RelinKey.

  Note GaloisKey is the type that equivalent to RelinKey.

Since a portion of the key is randomly generated, we also provide the compact operand to reduce the storage. For example, for a fresh ciphertext (c_0, c_1), the compact ciphertext operand store the seed of c_1 instead of c_1. The compact version of operand is shown in Figure 10 and they are consistent with the non-compact operands.

- **CompactCiphertext.** The CompactCiphertext operand is inherited from the OperandTuple template of Vector and CRTPolyNTT where the random part of a ciphertext is stored in a seed of Vector type. We provide the default construct method from configuration. That is, we can write as follows.

```
ckks::CompactCiphertext c;
```

To support the conversion between CompactCiphertext and Ciphertext, we define the Pack and UnPack operators.

```
//decompress a CompactCiphertext, i.e. generating the random part of
    ciphertext using stored seed.
ckks::Ciphertext ckks::Unpack(ckks::CompactCiphertext cc);

//Only for fresh symmetric encrypted ciphertext, the random part of a
    Ciphertext would be stored in a seed.
ckks::CompactCiphertxt ckks::Pack(ckks::Ciphertext c);
```

- **CompactKSKey, CompactRelinKeyFixedDnum and CompactRelinKey.** Similarly, the corresponding CompactXXXKey operands are also defined. Note that the Pack() and UnPack() operators are also given.

4.2.2 Operator Types of CKKS

We provide the basic homomorphic operators to perform the calculations, such as the basic modular arithmetic, encoding, encrypting, key switching, rotation etc. Note that we do not introduce the operators in detail and the reader can get more information through the code. Here we will only list some operators that need to be explained.

- **Encode/Decode.** The Encode and Decode operators for input Message/MessageComplex are provided as follows.

```
//Encode and Decode operator for Message.
ckks::Plaintext ckks::Encode(ckks::Message msg, Double scaleFactor);
ckks::Message ckks::Decode(ckks::Plaintext p);

//Encode and Decode operator for MessageComplex.
ckks::Plaintext ckks::EncodeComplex(ckks::MessageComplex msg,
                                    Double scaleFactor);
ckks::MessageComplex ckks::DecodeComplex(ckks::Plaintext p);
```

- **Encrypt and Decrypt.** Both symmetric and asymmetric encrypt operators are provided named as ckks:EncryptSymmetric() and ckks::EncryptAsymmetric(). For decrypting a Ciphertext, the ckks::Decrypt() is provided. Note that we also give the ckks::DecryptPublic() operators for the setting that the output can be obtained by a party besides the secret key owner.

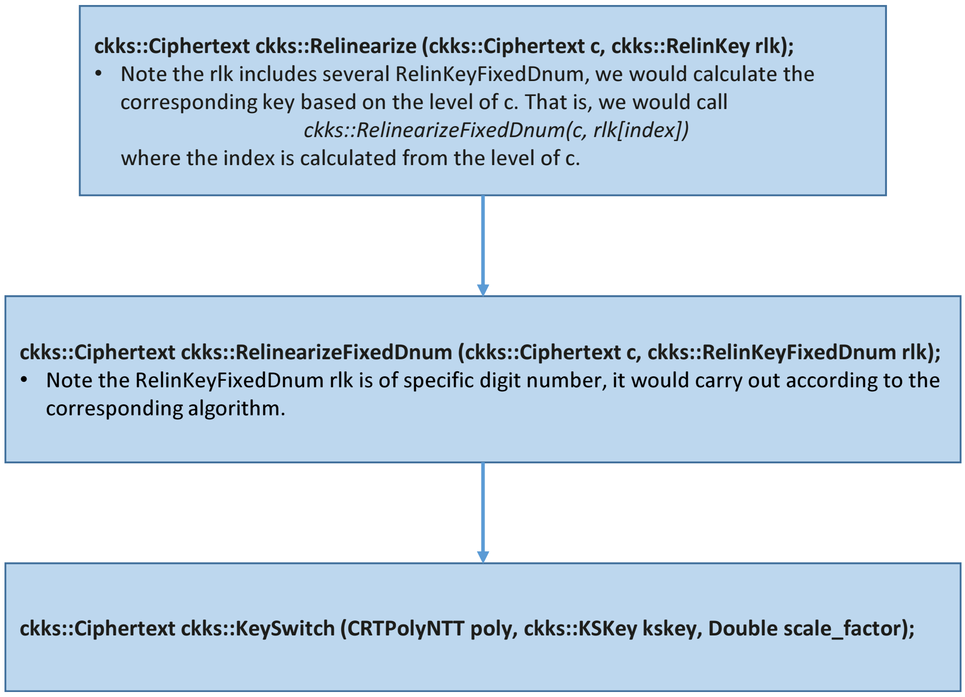
- **Key Generation.** We define the commonly key generation operators,such as ckks::KeyGenPublicKey(), ckks::KeyGenRelinKey(), ckks::KeyGenGaloisKey() etc. Note that we also provided the corresponding compact supporting versions, i.e. the generated key support the ckks::Pack() function.

- **Boosted Key Switching.** The KeySwitch operator takes the KSKey and the CRTPolyNTT as input, and make the key switching according to the information of the KSKey.

```
//key switching, note that the rns size of CRTPolyNTT should not be
    bigger than the kskey supported max level
ckks::Ciphertext ckks::KeySwitch(CRTPolyNTT poly, ckks::KSKey kskey,
                                 Double scale_factor);
```

The expansion function of KEY_SWITCHING operator would take the configuration of the KSKey(i.e. the digit number and the blocking information) to make the corresponding calculations.

- **Relinearization.**The user can take the key and ciphertext as inputs, and make the relinearization by call the ckks::Relinearize() operator. And its expansion is shown as follows.



ckks::Ciphertext ckks::Relinearize (ckks::Ciphertext c, ckks::RelinKey rlk);
• Note the rlk includes several RelinKeyFixedDnum, we would calculate the corresponding key based on the level of c. That is, we would call
ckks::RelinearizeFixedDnum(c, rlk[index])
where the index is calculated from the level of c.
ckks::Ciphertext ckks::RelinearizeFixedDnum (ckks::Ciphertext c, ckks::RelinKeyFixedDnum rlk);
• Note the RelinKeyFixedDnum rlk is of specific digit number, it would carry out according to the corresponding algorithm.
ckks::Ciphertext ckks::KeySwitch (CRTPolyNTT poly, ckks::KSKey kskey, Double scale_factor);

Figure 16: Relinear operators

- **Rotation.** We provide the rotation related operators as follows.

```
//the parameter step is a value that refers to a rotation step, a
    positive integer i indicates rotation to the left by i positions
    and a negative integer indicates a rotation of i positions to the
    right.
ckks::Ciphertext ckks::Rotate(ckks::Ciphertext c, ckks::GaloisKey glk,
                              int step);

//the conjugate rotation operator
ckks::Ciphertext ckks::RotateConjugate(ckks::Ciphertext c,
                                       ckks::GaloisKey glk);
```

Here we show an example to use the operators in CKKS. Assume we want to define the computing graph $x^2 + const * Rotate(x)$, the parameters and CKKS Context are well generated. Then we can write as follows.

```
Double scaleFactor = context->DefaultEncoderScaleFactor();
ckks::Message msg;
ckks::Message msgConst;
//the Encode operator support encoding of Message.
ckks::Plaintext p = ckks::Encode(msg, scaleFactor);
ckks::Plaintext pConst = ckks::Encode(msgConst, scaleFactor);

ckks::SecretKey sk;
```

```
ckks::PublicKey pk;
//the EncryptAsymmetric() support the asymmetric encryption.
ckks::Ciphertext ct0 = ckks::EncryptAsymmetric(pk, p);

ckks::RelinKey rlk;
ckks::GaloisKey glk;
//calculate the multiply of the ct0, then make relinearization and rescaling
ckks::Ciphertext ctSquare = ct0 * ct0 | ckks::RELIN(rlk) | ckks::Rescale;
//set the scale factor
ctSquare->SetScaleFactor(scaleFactor);

int step = 5;
//calculate the pconst * Rotate(x), then rescaling.
ckks::Ciphertext tmp = (pConst * ckks::Rotate(ct0, glk, step)) | ckks::
    Rescale;
tmp->SetScaleFactor(scaleFactor);
ckks::Ciphertext ct = tmp + ctSquare;
//decrypt and decode
ckks::Plaintext pt = ckks::Decrypt(ct, sk);
ckks::Message msg2 = ckks::Decode(pt);
```

4.3 User Interfaces

The calculation of FHE algorithms involves preparations such as parameter setting and key generation, we provide the user-friendly interfaces in ckks/user_interface.h.

4.3.1 The real data types

In the process of building a computational graph, we define the above operands that represent the placeholders instead of the real data. In ckks/user_interface.h, we provide the data types that store real data and allows users to define strongly typed application interfaces. They are all inherited from $std :: vector < uint64\_t >$ and shown in Fig. 17.

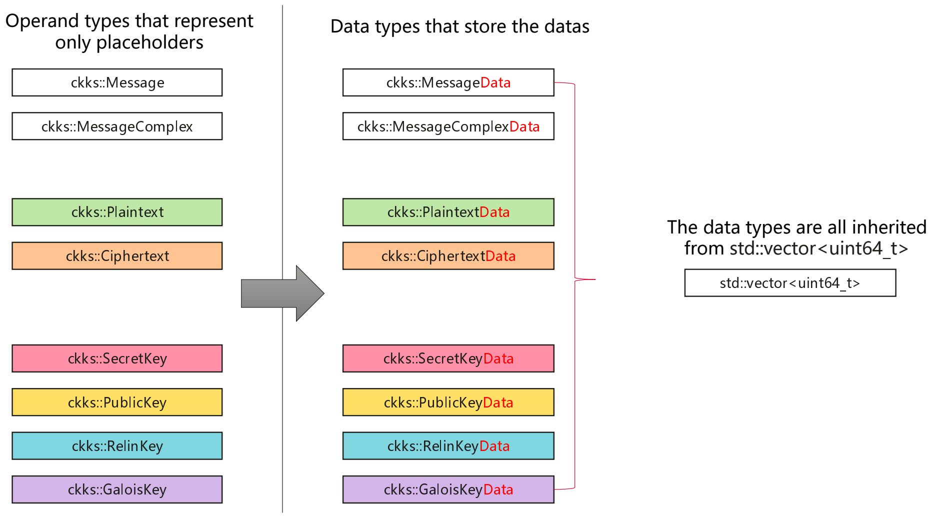


Figure 17: The real data types of CKKS algorithms

The xxxData types can be used to easily define interfaces. For example, in the KeyGenerator class, the data of relinear key, i.e. of RelinKeyData type can be generated by GetRelinKey() interface, then if we define a work that takes ckks::RelinKey as input, we can put the generated RelinKeyData object into the engine to run.

4.3.2 Parameter Generation

As for the parameters generation of CKKS algorithm, we provide three interfaces for various purposes. These interfaces return a pointer of the class CKKSParameters with different implementations.

- The GenerateCkksParameters function generates a group of parameters that the switching keys are same for ciphertexts of different levels. As shown in list 4.3.2, its input includes the following parameters.
- poly_degree, the polynomial degree, for example 1024,2048,...,16384 etc. Note that we only support the cyclotomic ring $\mathbf{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ currently, polynomial degree should be the power of 2. Support for more general cyclotomic rings is the future work.
- rns_mod_size, the vector of rns modulus bits, for example, $\{60, 40, 40\}$ refers to generate the ciphertext modulus with 60,40,40 bits prime numbers.
- extra_rns_mod, the vector of extra rns modulus bits that generate extra modulus used in key switching.
- encoder_scale, the scale factor of encoding, note that we define the default scale factor as 2^{40}.
- relinKeyFixedDnumSize, the size of relinear key, where the default choice is to generate a single key of encrypting s^2. If relinearization of $\{s^2, s^3, ..., s^k\}$ need to be supported, this parameter can be set to be $k-1$. And the default size is 1, i.e. only support relinearization of $\{s^2\}$
- rotate_steps, the vector of rotation steps that for generating corresponding keys. The default choice {} refers to generating a GaloisKey that supports arbitrary rotations by combining the fundamental rotations. If the specific keys needed to be generated, we can set the rotate_steps parameter as the step vector. For example, if we want to generate the rotation keys that corresponding to the steps $\{1, 4\}$ only, we can set the parameter as $\{1, 4\}$.

```
CKKSParametersPtr ckks::GenerateCkksParameters(uint32_t poly_degree,
  const std::vector<uint32_t>& rns_mod_size,
  const std::vector<uint32_t>& extra_rns_mod,
  double encoder_scale = DEFAULT_CKKS_ENCODER_SCALE_FACTOR,
  uint32_t relinKeyFixedDnumSize = DEFAULT_RELIN_KEY_FIXED_DNUM_SIZE,
  const std::vector<int64_t>& rotate_steps = {})
```

- In some scenarios, ciphertexts of different levels may use different switching keys. For example, in CraterLake [29], they use various dnum kskeys to achieve optimal overall performance. The GenerateCkksParametersDnum interface is provided to meet such requirements.

  It need a type of std::vector<std::tuple<uint32_t, uint32_t, std::vector$< uint32_t >>>$ to define dnum related parameters. Each element of the vector refers to the configuration of a KSKey. The first element in a tuple indicates the maximum ciphertext level supported by this KSKey, and the second parameter shows the dnum of the KSKey, and the last element refers to the KSKey's extra rns modulus.

```
CKKSParametersPtr ckks::GenerateCkksParametersDnum( uint32_t
    poly_degree,
  const std::vector<uint32_t>& rns_mod_size, const
  std::vector<std::tuple<uint32_t, uint32_t, std::vector<uint32_t>>>&
  blkConfInput, double encoder_scale =
      DEFAULT_CKKS_ENCODER_SCALE_FACTOR,
  uint32_t relinKeyFixedDnumSize = DEFAULT_RELIN_KEY_FIXED_DNUM_SIZE,
  const std::vector<int64_t>& rotate_steps = {})
```

We take an example to make a detailed interpretion. Let the polynomial degree be 16384, and the ciphertext modulus be $\{60, 40, 40, 40, 40, 40, 40\}$ where the max level is 6. If a KSKey with dnum 3 would be used for the ciphertext with level 6, and for ciphertext with level 5 and 4, a KSKey with dnum 2 need to be generated. Also there is a KSKey with dnum 1 for ciphertext with level ranging from 0 to 3, we can write the blkConfInput as follows:

```
auto blkConfInput = {{3,1,{60,60,61}}, {5,2,{60,60,21}},{6,3,{60,61}}
```

Note that three various dnum KSKey would be generated with different extra modulus, and they would be used for the ciphertexts with corresponding levels.

- As the configuration of KSKey is complex, we also provide the GenerateCkksParametersAuto function to make an automatic generation and more detailed implementation is shown in our code.

```
CKKSParametersPtr ckks::GenerateCkksParametersAuto(
  uint32_t poly_degree,
  const std::vector<uint32_t>& rns_mod_size,
  double encoder_scale = DEFAULT_CKKS_ENCODER_SCALE_FACTOR,
  uint32_t relinKeyFixedDnumSize = DEFAULT_RELIN_KEY_FIXED_DNUM_SIZE,
  const std::vector<int64_t>& rotate_steps = {})
```

4.3.3 Key Generation

We define a class KeyGenerator to generate the keys. It is constructed by CKKSParameters and provides common interfaces. For example, the GetSecretKey(), GetPublicKey(), GetRelinKey() function would return the data of secret key, public key, relinear key respectively.

Note that we also provide the compact way to store the keys that the interface's name is in the form of GetCompactXXXKey(). For example, the GetCompactPublicKey() interface would return a public key with its random part storing with a seed, instead of the random coefficients. The user can call this interface when saving data transmission and storage are needed.

4.3.4 Encoder

The encoding of messages is provided in the Encoder class. For encoding messages, it gives different interfaces to adapt to various message types, including the type of std::vector<double>, std::vector<std::complex<double>>, double etc. Note that if sparse encoding is needed, the SparseEncode() and SparseEncodeComplex() functions can be used.

For decoding the plaintext, the decode interface is similar to that of encoding and the user can select the corresponding function based on the type of the output message.

4.3.5 Encryptor

For encrypting a plaintext, the user can call the Encryptor class. If symmetric encryption would be used, it should construct the Encryptor by the secret key, while the public key would be used when constructing the Encryptor of asymmetric encryption.

For decrypting a ciphertext, our decrypt function support for various ciphertext size.

```
 PlaintextData Decrypt(const CiphertextData &ciphertext,
                       uint32_t ciphertext_size = 2);
```

For example, if the ciphertext is under encryption of s, s^2, s^3, then its size is 4 and we can write the decrypt function as follows.

```
PlaintextData pt = Decrypt(ciphertext, 4);
```

Note that [LM20] shows that if the output of the decryption algorithm can be obtained by a party besides the secret key owner, then it is possible to carry out a passive key recovery attack on CKKS. To avoid the attack, we provide the "DecryptPublic" interface to add a noise at the end of decryption, for further protection in such scenario.

```
PlaintextData DecryptPublic(const CiphertextData &ciphertext,
                            uint32_t error_bound_bits,
                            uint32_t ciphertext_size = 2);
```

The error_bound_bit parameter is the bounding bits of noise that added into the right plaintext and the user can set this parameter according to different requirements.

5 NTT and element-wise multiplication in HLG

In the section, we introduce our optimization of the calculation of NTT/INTT butterflies and element-wise multiplication in HLG. To perform efficient polynomial operations with higher degrees, we follow the state-of-the-art and implement Number theoretic transform (NTT), which has asymptotic complexity $O(n \log n)$.

Given a prime integer $q \equiv 1 \bmod 2n$ where n is a power of 2, let $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, $\boldsymbol{a} = (a[0], a[1], ..., a[n-1]) \in R_q$. We store the powers of $2n$-th primitive root of unity ψ ($\psi = \sqrt{\omega}$) $\in \mathbb{Z}_q$ as a look-up table Ψ_{rev}. All the coefficients of input and output polynomials are in standard order and bit-reversed order, respectively.

Let $\boldsymbol{a} = (a[0], a[1], ..., a[n-1])$, $\bar{\boldsymbol{a}} = (a[0], \psi a[1], ..., \psi^{n-1} a[n-1]) \in R_q$. To compute the polynomial multiplication $\boldsymbol{c} = \boldsymbol{ab} \in R_q$, we pre-compute and store the powers of $\psi^{-1} \in \mathbb{Z}_q$ as a look-up table Ψ^{-1}_{rev}, then output the negative wrapped convolution $\boldsymbol{c} = (1, \psi^{-1}, ..., \psi^{-(n-1)}) \circ \mathtt{NTT}^{-1}(\mathtt{NTT}(\bar{\boldsymbol{a}}) \circ \mathtt{NTT}(\bar{\boldsymbol{b}})) = \boldsymbol{ab} \in R_q$, where $\mathtt{NTT}^{-1}$ and $\circ$ denote the inverse of NTT and point-wise multiplication, respectively.

The related algorithms are shown in [28].

5.1 Lazy reduction

Let two non-negative integers a, b be congruent modulo a modulus q. If $a \in [0, q)$ is the reminder and $b > a$, we call the integer b lazy with respect to q. We have $b = (k-1)q + a \in \mathbb{Z}_{kq}$ where the integer $k \geq 1$.

An alternative efficient modular multiplication is the so-called Shoup method proposed by [31] if we knew one of multipliers which is smaller than the modulus in advance. We use Shoup method to calculate the lazy modular multiplication of butterflies, such that the intermediate data could be kept in lazy state. Hence, we will be able to remove all adjustments and branches of butterflies to further improve the performance of NTT/INTT.

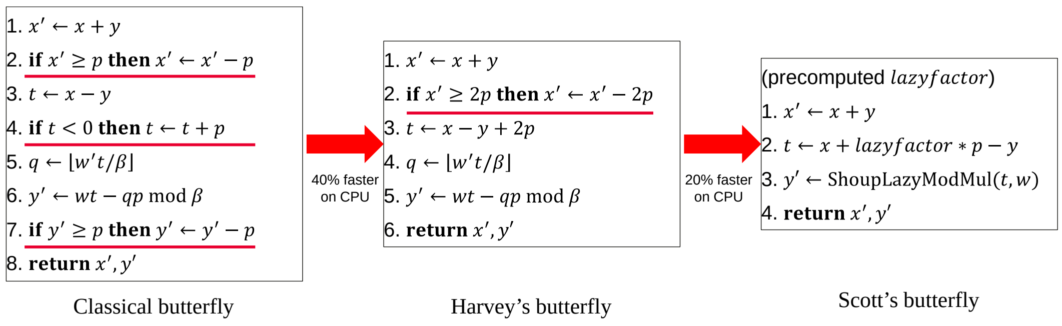
Let β be a positive integer, and $q < \beta/2$ be a modulus. We define a function $\mathtt{Quo}(a)$ that takes an integer $a \in \mathbb{Z}_q$ as input and outputs an integer equal to $\lfloor a\beta/q \rfloor$. In our case, the return value of $\mathtt{Quo}(a, q, \beta)$ is a 64-bit unsigned integer.

Definition 5.1 (Shoup method)**.** For two given integers $x \in \mathbb{Z}$ and $y \in \mathbb{Z}_q$, we define

$$\begin{aligned} r &= \mathtt{ShoupModMul}(x, y, q) \\ &= xy - q\lfloor x * \mathtt{Quo}(y, q, \beta)/\beta \rfloor \bmod \beta \\ &= xy \bmod 2q. \end{aligned}$$

Since the output of $\mathtt{Quo}(\psi, q, \beta)$ can be pre-computed where ψ is an element of the NTT/INTT LUTs, Shoup method is a good idea for the modular multiplications in NTT/INTT butterflies. In our case, we address delayed evaluation in NTT/INTT via

Shoup method. We follow an optimized method proposed by [30] which is based on Harvey's butterfly introduced in [23] and can remove all the branches from NTT/INTT butterflies. We don't use modular addtion/subtraction for NTT/INTT butterflies, which means that the increase of the value of the intermediate data will not be reduced until it may cause overflow.



```
1. $x' \leftarrow x + y$
2. if $x' \geq p$ then $x' \leftarrow x' - p$
3. $t \leftarrow x - y$
4. if $t < 0$ then $t \leftarrow t + p$
5. $q \leftarrow \lfloor w't/\beta \rfloor$
6. $y' \leftarrow wt - qp \bmod \beta$
7. if $y' \geq p$ then $y' \leftarrow y' - p$
8. return $x', y'$
```

```
1. $x' \leftarrow x + y$
2. if $x' \geq 2p$ then $x' \leftarrow x' - 2p$
3. $t \leftarrow x - y + 2p$
4. $q \leftarrow \lfloor w't/\beta \rfloor$
5. $y' \leftarrow wt - qp \bmod \beta$
6. return $x', y'$
```

```
(precomputed $lazyfactor$)
1. $x' \leftarrow x + y$
2. $t \leftarrow x + lazyfactor * p - y$
3. $y' \leftarrow$ ShoupLazyModMul$(t, w)$
4. return $x', y'$
```

Figure 18: By following Scott's optimized method, the last branch in Harvey's butterfly can be also removed

It should be noted that the NTT butterfly computes addtion/subtraction after multiplication, that is, lazy reduction, whereas the INTT butterfly has the opposite process. Therefore, the optimization of lazy evaluation for INTT will be more complicated.

5.2 Montgomery modular multiplication

Montgomery modular multiplication is an efficient method for large integer multiplication. Given two non-negative integers x, y and a modulus q, Montgomery modular multiplication will output a q-residue that equal to $xyr^{-1} \mod q$, where the auxiliary parameter $r \in \mathbb{Z}$ is larger than q and satisfies $\gcd(r, q) = 1$.

Definition 5.2 (Montgomery form and Montgomery reduction)**.** For a given non-negative integer $a \in \mathbb{Z}_q$, we define Montgomery reduction that inputs an integer $a \in \mathbb{Z}_q$ and outputs an integer that equal to $ar^{-1} \mod q$. We also define by $a' = ar \mod q$ the Montgomery form of a.

In our case, a non-negative integer which is equal or congruent to a', for instance, $a' + kq$ $(k \geq 0)$, can also be considered as in Montgomery form of a, because Montgomery reduction will output the same result if inputting different values which has congruence relation.

The mapping $m(x) : x \in \mathbb{Z}_q \to x' \in \mathbb{Z}_q$ is a bijection and a field isomorphism of $\mathbb{Z}_q$. We can also convert such integer between ordinary domain and Montgomery domain via Montgomery reduction. We implemented the radix-2^ω Montgomery modular multiplication introduced in [33] by splitting an arbitrary n-bit multiplier into $s = \lfloor n/\omega \rfloor$ blocks.

In conclusion, we perform Shoup method to address lazy reduction for NTT/INTT butterflies, and hence all branches in butterflies can be removed. Our NTT operator outputs a polynomial whose coefficients are in Montgomery form, such that we can compute the element-wise multiplication via Montgomery modular multiplication.

6 Benchmark

For now, HLG only supports multi-CPU compute engine as back end. Here we only provide benchmark results for both HLG and SEAL v3 in multi-threading scenarios.

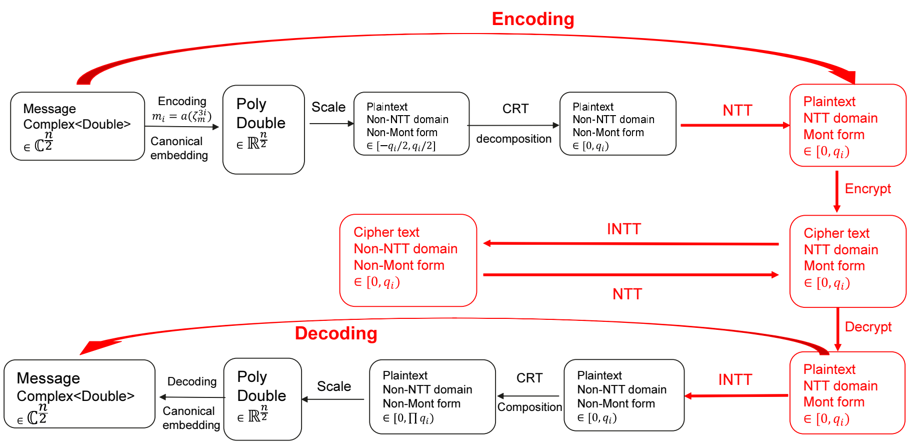

Encoding
Message Complex<Double> $\in \mathbb{C}^{\frac{n}{2}}$
Encoding $m_i = a(\zeta_m^{3i})$
Canonical embedding
Poly Double $\in \mathbb{R}^{\frac{n}{2}}$
Scale
Plaintext Non-NTT domain Non-Mont form $\in [-q_i/2, q_i/2]$
CRT decomposition
Plaintext Non-NTT domain Non-Mont form $\in [0, q_i)$
NTT
Plaintext NTT domain Mont form $\in [0, q_i)$
Encrypt
Cipher text Non-NTT domain Non-Mont form $\in [0, q_i)$
INTT
NTT
Cipher text NTT domain Mont form $\in [0, q_i)$
Decrypt
Decoding
Message Complex<Double> $\in \mathbb{C}^{\frac{n}{2}}$
Decoding
Canonical embedding
Poly Double $\in \mathbb{R}^{\frac{n}{2}}$
Scale
Plaintext Non-NTT domain Non-Mont form $\in [0, \prod q_i)$
CRT Composition
Plaintext Non-NTT domain Non-Mont form $\in [0, q_i)$
INTT
Plaintext NTT domain Mont form $\in [0, q_i)$

Figure 19: The data states of CKKS implementation in HLG

6.1 Example of execution trace of multi-CPU compute engine of HLG

In our current implementation, the computation of ciphertexts of CKKS only need 8 primitive operators: op_permutation_ntt(Galois automorphism), op_add, op_neg, op_mul, op_scalar_mul, op_ntt, op_intt, op_mod_switch.

Fig. 20 shows the execution trace of HLG's MyEngine with 7 threads. Note that the colored block indicates primitive operators and the white gaps are the overhead of task scheduling.

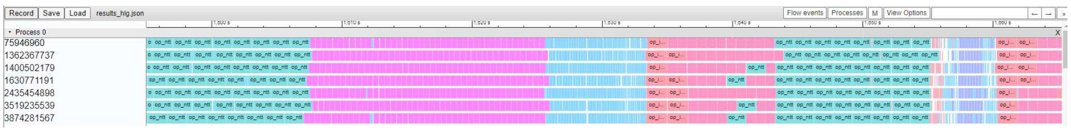


Figure 20: Example of execution trace of multi-CPU compute engine

It is easy to find that HLG framework can use multiple threads very efficiently: none of the threads stay idle for a very long time. This is because when we expand the operators to RNSPoly level, we can fully utilize the natural parallelism provided by RNS system.

6.2 Test cases for benchmark

6.2.1 Case with high parallelism: Digit recognition on MNIST data set

The first test case is digit recognition on ciphertext. The data set is MNIST. The solution we implemented is Lola from [8]. Note that this test case has very high parallelism in its shape of computing graph as shown in Fig. 21.

The comparison of performance of Lola with SEAL and HLG is shown in Fig. 22, where MyEngine is used for HLG.

Note that HLG is slower than SEAL with 1 thread only. The reason is that current implementation of multi-CPU compute engine has overhead on dynamic task scheduling. When the number of threads increases, HLG's performance becomes better than SEAL. This is because that HLG allows operator expansion to a smaller granularity, while seal can only support manual task scheduling at the level of ciphertext/plaintext interfaces by hand. As a result, HLG can use multiple cpu cores and threads more effectively. In this case, with 20 threads, HLG can be 24.4% faster than SEAL.

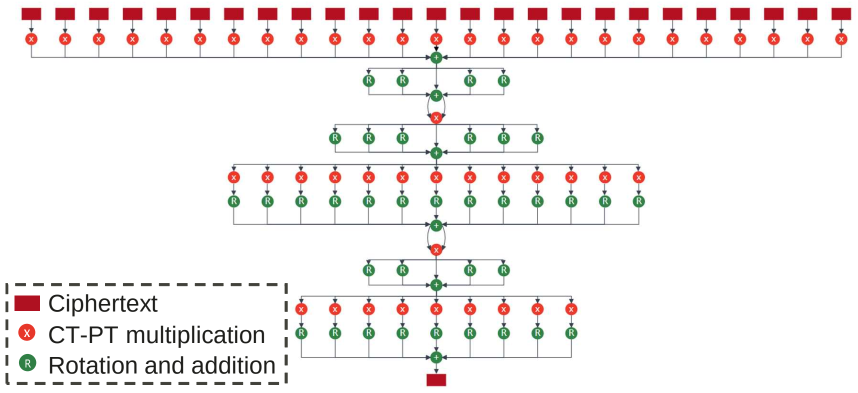


Figure 21: Shape of computing graph of Lola

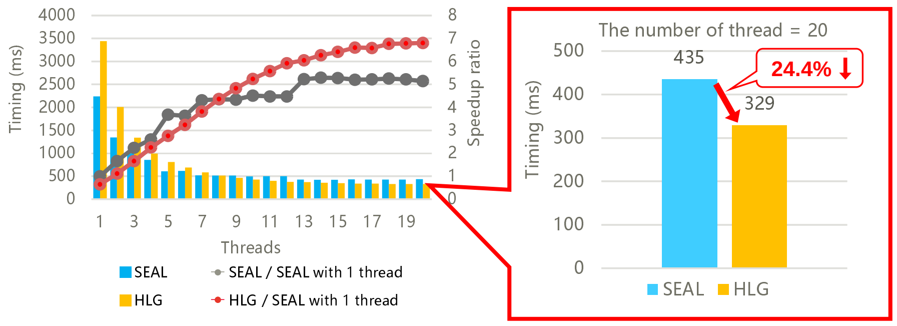


Figure 22: Comparison of performance of Lola with SEAL and HLG

6.2.2 Case with low parallelism: Calculating square root

The second test case is calculating square root on ciphertext. We choose an iterated algorithm from [10], which only supports input value in $[0, 1]$ as in Fig. 23.

The comparison of performance of SQRT with SEAL and HLG is shown in Fig. 24, where MyEngine is used for HLG.

In this case, HLG is much faster than SEAL, up to 85.6%. Since SEAL only provides interfaces at ciphertext/plaintext level, we can use at most 3 threads by hand. Since HLG can support auto parallel task scheduling at RNSPoly level, we can still use multiple threads very efficiently.

6.3 Other cases where HLG is slower than SEAL

We have also tested other cases where HLG is slower than SEAL.

- The first example is 32 parallel rotation of ciphertexts. In this case, the computing graph is perfect for manual task scheduling for SEAL, while HLG still has overhead for dynamic task scheduling at the level of RNSPoly.

- The second example is matrix multiplication where the matrix is plaintext while the vector is ciphertext. In this case, all primitive operators are scalar multiplications, which is much cheaper than NTT, Montgomery Multiplication and other operators. As a result, the proportion of the overhead of task scheduling is significantly higher than that of conventional computing tasks.

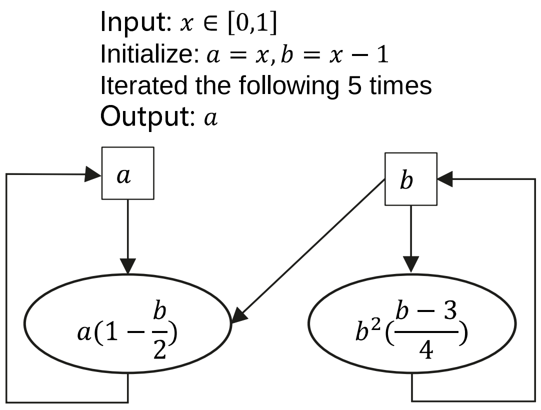


Figure 23: Shape of computing graph of iterated SQRT algorithm

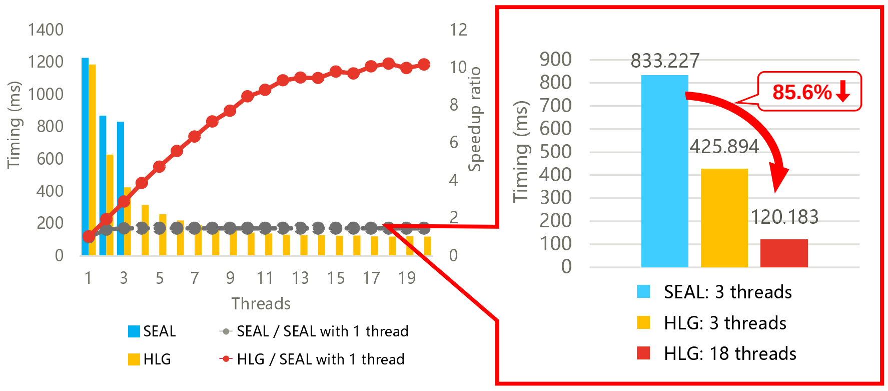


Figure 24: Comparison of performance of SQRT with SEAL and HLG

There are multiple ways for optimization: we can either change the level of primitive operator to higher level (such as ciphertext/plaintext level), or develop a better compiling strategy for execution with 1 thread only. Both requires extension of current HLG framework in multiple aspects. We will consider these as part of the future plan.

6.4 An observation on the performance bottleneck with multiple CPU cores

During of tests and experiments of HLG in different use cases, we had an interesting observation on the acceleration ratio of HLG's multi-CPU compute engines with different number of threads on different hardware platforms. We believe we have confirmed the performance bottleneck of FHE with multiple CPU cores: the memory IO bandwidth.

In the beginning, we had a confusing observation: The acceleration ratio never exceeds 10 on our linux server, no matter how many threads we use with MyEngine. The first guess is that maybe this is because we have used too many mutexs.

In order to verify the initial idea, we tried to implement LockFreeEngine, where no mutex is used. However, the problem still exists: the curve of acceleration ratio of MyEngine and LockFreeEngine are very similar as the number of threads increases. Both of the two curves approach the acceleration ratio of 10 times and never goes beyond that. It is not about the mutex.

We looked at the benchmark results of the primitive operators, which shows that the time cost of NttRns and InttRns operators increases dramatically when we use more threads in both MyEngine and LockFreeEngine. This is even more mind-boggling.

In order to find out why this happens, we decided to implement the same test case using SEAL with multiple threads. Then we found a similar curve as HLG, where acceleration ratio never exceeds 10. The curves of acceleration ratio of MyEngine, LockFreeEngine and SEAL is shown in Fig. 25.

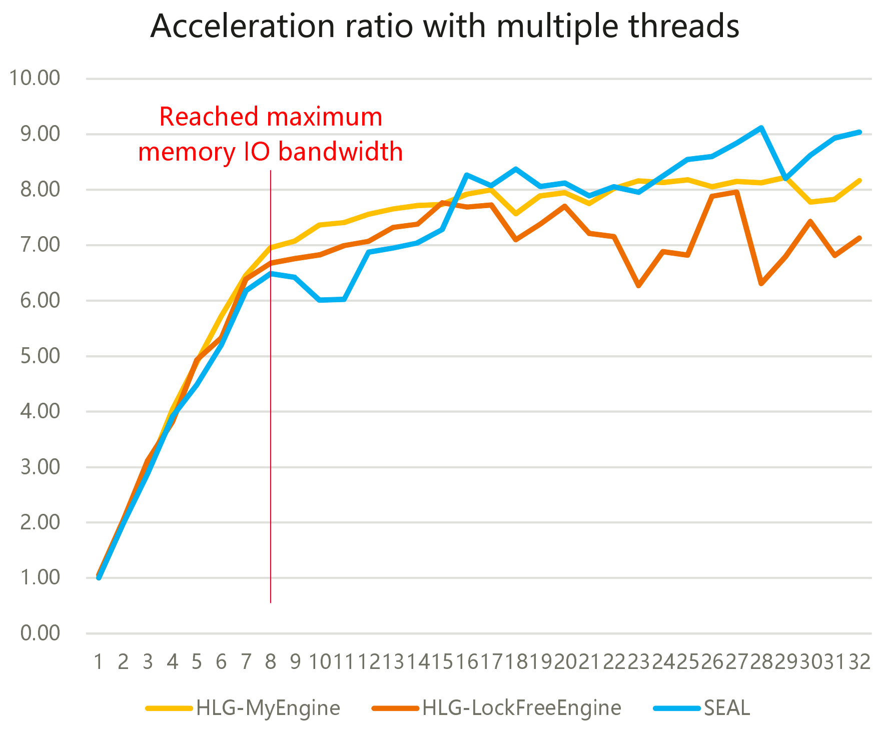


Figure 25: Comparison of acceleration ratio of MyEngine, LockFreeEngine and SEAL with different number of threads

By observing these curves, we can easily find that when the number of threads is small, overall performance increases linearly with the number of threads.

Note 1: the maximum acceleration ratio and the threshold number of threads may vary for different use cases.

Note 2: the value of the acceleration ratio becomes unpredictable after reaching the limitation. The curves will look random every time we run the tests.

Now we can have a conclusion that when we use more than 8 threads, we have reached the maximum memory IO bandwidth of our linux server. The linux server's parameters are: X86 instruction set, L1 cache size 16KB per core, 512GB memory.

To further verify the conclusion, we switch to another server with openEuler OS and KunPeng 920 CPU. Then the maximum acceleration ratio increases to about 16 with about 19 threads on this new hardware platform. Detailed parameters of this new platform are: ARM instruction set, L1 cache 64KB per core, 512GB memory. It seems that larger L1 cache size and higher memory IO bandwidth indeed contributes to the overall end-to-end performance of both HLG and SEAL.

The reason why NttRns and InttRns operators becomes slower is also because: when running with multiple threads, many threads need to access the memory at the same time and it is easier to reach the maximum IO bandwidth. In this case, Ntt/Intt operators with large polynomial degrees will become slower since they require a lot of data access beyond L1 cache, to L2/L3 cache or even to the memory.

6.5 HEBench results?

In HEBench framework [2] all test cases are implemented with hard-coded task scheduling strategy. This may be constraint to the ability of HLG framework which allows customized task scheduling strategies.

The other problem is that benchmark of some unit tests does not lead to reliable estimation of end-to-end performance for some complicated use cases. We may need more end-to-end test cases instead of unit tests like matrix multiplications only.

Thus we propose to use a more abstract way to describe the test cases, instead of hard-coded framework with only static task scheduling with 1 thread. For example, we can describe a test case as a computing graph with HLG IR and save it as a binary work file. Then the engineers are free to choose any task scheduling strategies they want for overall optimization.

6.6 Benchmark results for other hardware platforms

[TBD] We don't support other hardware platforms yet!

7 Conclusion and Future Plan

In this paper, we described HLG, a framework for RNS system supporting decoupling of front-end algebraic algorithm and back-end hardware. The core of HLG framework is HLG IR, which allows splitting and combination of data placeholders and make the development of RNS algorithm easier.

To sum it up:

- Basic functionalities and design rationale of HLG framework and HLG IR
- Details of customized operand and operator types based on HLG at the level of RLWE and CKKS
- Details of implementation of primitive operators
- Benchmark of HLG-CKKS and comparison to SEAL v3.

So far, we have verified the effectiveness of HLG framework and HLG IR for the RNS system by exploring its application in researches and implementation of FHE algorithm.

In the future, there are multiple possibilities to improve HLG framework, including but not limited to:

- Changing HLG's front-end from C++ to python and support dynamic graph generation, in order to support better experience for debugging.

  Note: now HLG shares the same weakness as Tensorflow 1.0 since they both supports static graph only.
- Migrating HLG's customized algebraic IR to MLIR. Need to solve the problem of how to express computing graphs in RNS, especially, how to support combination and splitting of placeholders.
- Supporting more FHE algorithms. Such as BFV, BGV, TFHE and other new algorithms.

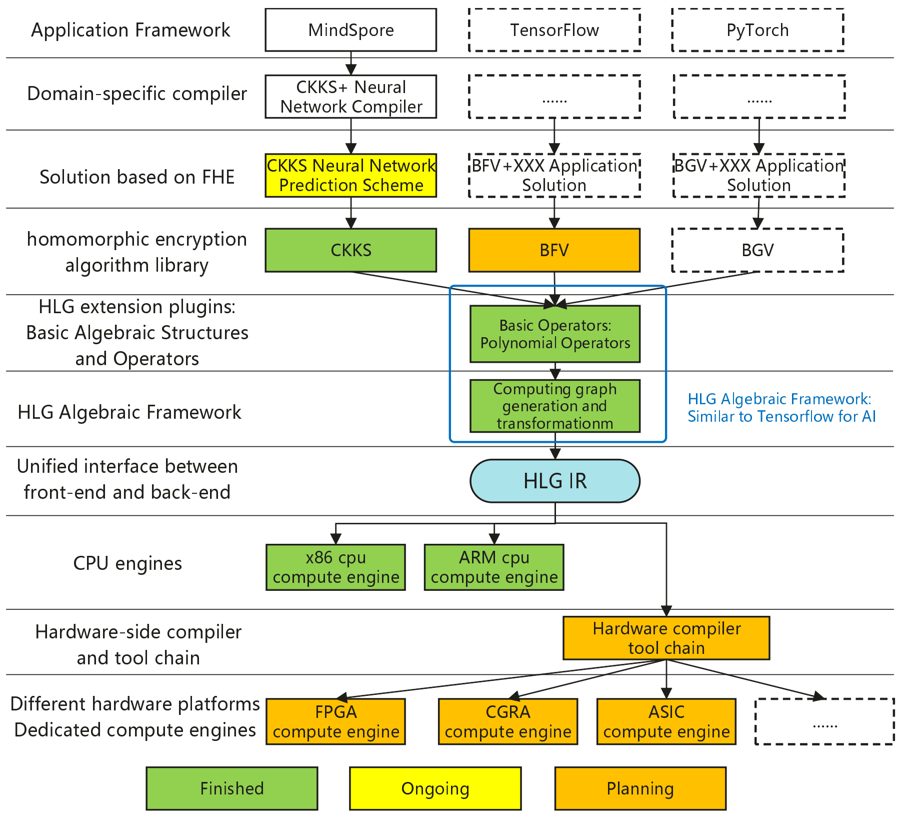

Application Framework
MindSpore
TensorFlow
PyTorch
Domain-specific compiler
CKKS+ Neural Network Compiler
......
......
Solution based on FHE
CKKS Neural Network Prediction Scheme
BFV+XXX Application Solution
BGV+XXX Application Solution
homomorphic encryption algorithm library
CKKS
BFV
BGV
HLG extension plugins: Basic Algebraic Structures and Operators
Basic Operators: Polynomial Operators
HLG Algebraic Framework
Computing graph generation and transformationm
HLG Algebraic Framework: Similar to Tensorflow for AI
Unified interface between front-end and back-end
HLG IR
CPU engines
x86 cpu compute engine
ARM cpu compute engine
Hardware-side compiler and tool chain
Hardware compiler tool chain
Different hardware platforms Dedicated compute engines
FPGA compute engine
CGRA compute engine
ASIC compute engine
......
Finished
Ongoing
Planning

Figure 26: Our vision for the technology stack of HLG in the future

- Supporting for more different hardware platforms. Such as GPU, FPGA, AI chips, etc.
- Hardware/application-level compiler for better programmability and optimization.
- Middleware solutions for variant use cases based on HLG. Such as Linear/Logistic regression training, XGBoost federated learning, etc.
- Extension to support MPC operators and even more. To form a unified development tool for Hybrid solution of FHE and MPC, with similar experience as developing local application.
- Integration into high-level application framework such as AI frameworks: MindSpore, Tensorflow, PyTorch, etc.

Our ultimate goal is to form a complete software stack from hardware to application as shown in Fig. 26. Since HLG framework can play a similar role as Tensorflow for AI, we would like to collaborate with both academia and industry to explore and improve HLG framework continuously.

Thanks

Thanks to our ex-colleague Tianyi Ai and our intern Zhengyu Shi who have made decent contribution to the implementation of compute engine and boosted key switch in HLG framework.

References

[1] https://www.darpa.mil/news-events/2021-03-08

[2] Homomorphic encryption benchmarking framework - hebench (release v0.9). Online: https://github.com/hebench/frontend (Nov 2022), algorand Foundation, CryptoLab, Deloitte, Duality, IBM Research, Intel, KU Leuven, Microsoft Research, Tune Insight SA

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI. vol. 16, pp. 265–283 (2016)

[4] Badawi, A.A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., Saraswathy, R.V., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: Openfhe: Open-source fully homomorphic encryption library. IACR Cryptol. ePrint Arch. p. 915 (2022), https://eprint.iacr.org/2022/915

[5] Bajard, J.C., Eynard, J., Hasan, M.A., Zucca, V.: A full rns variant of fv like somewhat homomorphic encryption schemes. In: International Conference on Selected Areas in Cryptography. pp. 423–442. Springer (2016)

[6] Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Annual Cryptology Conference. pp. 868–886. Springer (2012)

[7] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6(3), 1–36 (2014)

[8] Brutzkus, A., Elisha, O., Gilad-Bachrach, R.: Low latency privacy preserving inference. CoRR abs/1812.10659 (2018), http://arxiv.org/abs/1812.10659

[9] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017)

[10] Cheon, J.H., Kim, D., Kim, D., Lee, H.H., Lee, K.: Numerical method for comparison on homomorphically encrypted numbers. Cryptology ePrint Archive, Paper 2019/417 (2019), https://eprint.iacr.org/2019/417, https://eprint.iacr.org/2019/417

[11] Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: international conference on the theory and application of cryptology and information security. pp. 3–33. Springer (2016)

[12] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 377–408. Springer (2017)

[13] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Tfhe: fast fully homomorphic encryption over the torus. Journal of Cryptology 33(1), 34–91 (2020)

[14] Chillotti, I., Joye, M., Paillier, P.: Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In: International Symposium on Cyber Security Cryptography and Machine Learning. pp. 1–19. Springer (2021)

[15] Ducas, L., Micciancio, D.: Fhew: bootstrapping homomorphic encryption in less than a second. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 617–640. Springer (2015)

[16] Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. 2012, 144 (2012)

[17] Gailly, J., Adler, M.: zlib compression library (2022), https://zlib.net/

[18] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)

[19] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer (2012), https://doi.org/10.1007/978-3-642-32009-5_49

[20] Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Annual Cryptology Conference. pp. 75–92. Springer (2013)

[21] Halevi, S., Polyakov, Y., Shoup, V.: An improved RNS variant of the BFV homomorphic encryption scheme. In: Matsui, M. (ed.) Topics in Cryptology - CT-RSA 2019 - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11405, pp. 83–105. Springer (2019), https://doi.org/10.1007/978-3-030-12612-4_5

[22] Han, K., Ki, D.: Better bootstrapping for approximate homomorphic encryption. In: Jarecki, S. (ed.) Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12006, pp. 364–390. Springer (2020), https://doi.org/10.1007/978-3-030-40186-3_16

[23] Harvey, D.: Faster arithmetic for number-theoretic transforms. Journal of Symbolic Computation 60, 113–119 (2014)

[24] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis and transformation. In: CGO '04: Proceedings of the international symposium on Code generation and optimization. p. 75. IEEE Computer Society, Washington, DC, USA (2004), http://portal.acm.org/citation.cfm?id=977395.977673&coll=GUIDE&dl=GUIDE&CFID=48424181&CFTOKEN=16724426

[25] Lattner, C., Pienaar, J.A., Amini, M., Bondhugula, U., Riddle, R., Cohen, A., Shpeisman, T., Davis, A., Vasilache, N., Zinenko, O.: MLIR: A compiler infrastructure for the end of moore's law. CoRR abs/2002.11054 (2020), https://arxiv.org/abs/2002.11054

[26] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)

[27] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., K?pf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library (2019), http://arxiv.org/abs/1912.01703, cite arxiv:1912.01703Comment: 12 pages, 3 figures, NeurIPS 2019

[28] Pöppelmann, T., Oder, T., Güneysu, T.: High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In: Cryptology - LATINCRYPT 2015. vol. 9230, pp. 346–365 (2015)

[29] Samardzic, N., Feldmann, A., Krastev, A., Manohar, N., Genise, N., Devadas, S., Eldefrawy, K., Peikert, C., Sánchez, D.: Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In: Salapura, V., Zahran, M., Chong, F., Tang, L. (eds.) ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022. pp. 173–187. ACM (2022), https://doi.org/10.1145/3470496.3527393

[30] Scott, M.: A note on the implementation of the number theoretic transform. IACR Cryptol. ePrint Arch. p. 727 (2017), https://eprint.iacr.org/2017/727

[31] Shoup, V.: Ntl: A library for doing number, http://www.shoup.net/ntl/

[32] Varda, K.: Protocol buffers: Google's data interchange format. Tech. rep., Google (6 2008), http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html

[33] Yuan, Y., Fukushima, K., Xiao, J., Kiyomoto, S., Takagi, T.: Memory-constrained implementation of lattice-based encryption scheme on standard java card platform. IET Information Security 15(4), 267–281 (2020)