Accelerating Encrypted Computing on Intel GPUs

Yujia Zhai,* Yiqin Qiu,† Alexey Titov,† Alexander Lyashevsky,†
*University of California, Riverside, CA, USA
†Intel Corporation, Santa Clara, CA, USA
yujiazhai94@gmail.com, yiqin.qiu@intel.com, alexey.titov@intel.com, alexander.lyashevsky@intel.com

***Index Terms*—Homomorphic Encryption, Number Theoretic Transform, Intel GPU, CKKS, Privacy-Preserving Computing**

I. Introduction

Cloud computing has a growing demand and it was only boosted in the last two years. By 2021, 50% of enterprise workloads were deployed to public clouds, and this percentage is expected to reach 57% in the next 12 months [1]. Although outsourcing data processing to cloud resources enables enterprises to relieve the overhead of deployment and maintenance for their private servers, it raises security and privacy concerns of the potential sensitive data exposure.

Adopting traditional encryption schemes to address this privacy concern is less favorable because a traditional encryption scheme requires decrypting the data before the computation, which presents a vulnerability and may destroy the data privacy. In contrast, Homomorphic Encryption (HE), an emerging cryptographic encryption and computation scheme, is considered to be one of the most promising solutions to such issues. HE allows computations to be performed directly on encrypted messages without the need for decryption. This encryption scheme, thus, protects private data from both internal malicious actors and external intruders, while assuming honest computations.

In 1978, Rivest, Adleman, and Dertouzous [2], first introduced the idea of computing on encrypted data through the use of "privacy homomorphisms". Since then, several distinct HE schemes have been invented, which can be categorized by the types of encrypted computation they support. *Partial* HE schemes enable only encrypted additions or multiplications. The famous RSA cryptosystem is, in fact, the first HE scheme, supporting encrypted modular multiplications. In contrast, the Paillier cryptosystem [3] is a partial HE scheme that supports only modular additions.

Levelled HE schemes, on the other hand, support both encrypted additions and multiplications, but only up to a certain circuit depth determined by the encryption parameters. The Brakerski/Fan-Vercauteren (BFV) [4] and Brakerski-Gentry-Vaikuntanathan (BGV) [5] schemes are two popular leveled HE schemes used today, which support exact integer computation. In [6], Cheon, Kim, Kim and Song presented the CKKS scheme, which treats the encryption noise as part of approximation errors that occur during computations within floating-point numerical representation. This imprecision requires a refined security model [7], but provides faster runtimes than BFV/BGV in practice.

Fully HE schemes enable an unlimited number of encrypted operations, typically by adding an expensive bootstrapping step to a levelled HE scheme, as first detailed by Craig Gentry [8]. TFHE [9] improves the runtime of bootstrapping, but requires evaluating circuits on binary gates, which becomes expensive for standard 32-bit or 64-bit arithmetic. The improved capabilities and performance of these HE schemes have enabled a host of increasingly sophisticated real-world privacy-preserving applications. Early applications included basic statistics and logistic regression evaluation [10]. More recently, HE applications have expanded to a wide variety of applications, including privatized medical data analytics and privacy-preserving machine learning [11]–[15].

To address the memory and runtime overhead of HE — a major obstacle to immediate real-world deployments, HE libraries support efficient implementations of multiple HE schemes, including Microsoft SEAL [16] (BFV/CKKS), HElib [17] (BFV/BGV/CKKS), and PALISADE [18] (BGV/BFV/CKKS/TFHE). In [19], Intel published HEXL, accelerating HE integer arithmetic on finite fields by featuring Intel Advanced Vector Extensions 512® (Intel AVX512) instructions. Since GPUs deliver higher memory bandwidth and computing throughput with lower normalized power consumption, researchers presented libraries such as cuHE [20], TFHE [9] and NuFHE [21] to accelerate HE using CUDA-enabled GPUs.

Although HE optimizations on CPUs and CUDA-enabled GPUs have been reported before, an architecture-aware HE library optimized for Intel GPUs has not been available. In addition, previous works that accelerate HE libraries majorly focus on optimizing Number Theoretic Transform (NTT) and inverse NTT (iNTT) computing kernels, since these two algorithms account for substantial execution time of HE routines (e.g., 72%-81% in its baseline variant on Intel GPUs according to our benchmarks in Fig. 9). However, engineering an efficient HE library requires systematical optimizations for the whole HE pipeline beyond computing kernels. In this chapter, we present a HE library optimized for Intel GPUs based on the CKKS scheme. We not only provide a set of highly optimized computing kernels such as NTT and iNTT, but also optimize the whole HE evaluation pipeline at both the instruction level and application level. More specifically, these are the topics that we are discussing in this chapter:

- We design and develop the first-ever SYCL-based GPU backend for Microsoft SEAL APIs, which is also the first HE library based on the CKKS scheme optimized for

Intel GPUs.

- We develop and discuss a staged implementation of NTT leveraging shared local memory of Intel GPUs. We also describe NTT optimization by employing strategies including high-radix algorithm, kernel fusion, and explicit multiple-tile submission.
- From the instruction level perspective, we describe how to enable low-level optimizations for 64-bit integer modular addition and modular multiplication using inline assembly. We also provide a fused modular multiplication-addition operation to reduce the number of costly modular operations.
- From the application level, we introduce the memory cache mechanism to recycle freed memory buffers on device to avoid the run-time memory allocation overhead. We also design fully asynchronous HE operators and asynchronous end-to-end HE evaluation pipelines.
- We benchmark our HE library on two latest Intel GPUs. Experimental results show that our NTT implementations reaches up to 79.8% and 85.7% of the theoretical peak performance on both experimental GPUs, faster than the naive GPU baseline by 9.93X and 7.02X, respectively.
- The proposed and thoroughly discussed NTT and assembly-level optimizations accelerate five HE evaluation routines under the CKKS scheme by 2.32X - 3.05X. In addition, the polynomial element-wise matrix multiplication applications are accelerated by 2.68X - 3.11X by our all-together systematic optimizations.

The rest of the chapter is organized as follows: we introduce background and related works in Section II. We then detail the system-level designs and optimizations in Section III. We present the optimization approaches in Section V as well as other development strategies in Section IV. Evaluation results are provided in Section VI. We conclude our chapter and present potential future work in Section VII.

II. Background and Related Works

In this section, we briefly introduce the basics of the CKKS HE scheme. We then introduce the general architecture of Intel GPUs and summarize prior works of NTT optimizations on both CPUs and GPUs.

A. Basics of CKKS

The CKKS scheme was first introduced in [6], enabling approximation computation on complex numbers. This approximate computation is particularly suitable for real-world floating-point operations that are approximate by design. Further work improved CKKS to support a full residue number system (RNS) [22] and bootstrapping [23]. In this chapter, we select CKKS as our FHE scheme, as implemented in Microsoft SEAL [16].

The CKKS scheme is composed of following basic primitives: *KeyGen*, *Encode*, *Decode*, *Encrypt*, *Decrypt*, *Add*, *Multiply* (*Mul*), *Relinearize* (*Relin*) and *Rescale* (*RS*). To be more specific, *KeyGen* first generates a set of keys for the CKKS scheme. An input message is encoded to a plaintext and then encrypted to a ciphertext. One can evaluate (compute) directly on the encrypted messages (ciphertexts). Noises are accumulated during the HE evaluation until one applies a *Relin* followed by a *RS* to the ciphertext. Once all the HE computations are completed, the result ciphertext is decrypted and decoded, providing the same result as ordinary non-HE computations. We provide only cursory descriptions here and refer interested readers to [6] for details.

B. Number Theoretic Transform and Residue Number System

As noted in [24], the NTT can be exploited to accelerate multiplications in the polynomial ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^N+1)$. We represent polynomials using a coefficient embedding: $\mathbf{a} = (a_0, ..., a_{N-1}) \in \mathbb{Z}_q^N$ and $\mathbf{b} = (b_0, ..., b_{N-1}) \in \mathbb{Z}_q^N$. Let ω be a primitive N-th root of unity in $\mathbb{Z}_q$ such that $\omega^N \equiv 1(\mod q)$. In addition, let ψ be the $2N$-th root of unity in $\mathbb{Z}_q$ such that $\psi^2 = \omega$. Further defining $\tilde{\mathbf{a}} = (a_0, \psi a_1, ..., \psi^{N-1} a_{N-1})$ and $\tilde{\mathbf{b}} = (b_0, \psi b_1, ..., \psi^{N-1} b_{N-1})$, one can quickly verify that for $\mathbf{c} = \mathbf{a} \cdot \mathbf{b} \in \mathbb{Z}_q^N$, there holds the relationship $\mathbf{c} = \mathbf{\Psi^{-1}} \odot \text{iNTT}(\text{NTT}(\tilde{\mathbf{a}}) \odot \text{NTT}(\tilde{\mathbf{b}}))$. Here $\odot$ denotes element-wise multiplication and $\mathbf{\Psi^{-1}}$ represents the vector $(1, \psi^{-1}, \psi^{-2}, ..., \psi^{-(N-1)})$. Therefore, the total computational complexity of ciphertext multiplication in $\mathcal{R}_q$ is reduced from $O(N^2)$ to $O(N \log N)$.

In practice, since polynomial coefficients in the ring space are big integers under modulus q, multiplying these coefficients becomes computationally expensive. The Chinese Remainder Theorem (CRT) is typically employed to reduce this cost by transforming large integers to the Residue Number System (RNS) representation. According to CRT, one can represent the large integer $x \mod q$ using its remainders $(x \mod p_1, x \mod p_2, \ldots, x \mod p_n)$, where the moduli $(p_1, p_2, ..., p_n)$ are co-prime such that $\Pi p_i = q$. We note the CKKS scheme has been improved from the initial presentation in Section II-A to take full advantage of the RNS [22].

To summarize what we have discussed, to multiply polynomials $\mathbf{a}$ and $\mathbf{b}$ represented as vectors in $\mathbb{Z}_q^N$, one needs to first perform the NTT to transform the negative wrapped $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$ to the NTT domain. After finishing element-wise polynomial multiplication in the NTT domain, the iNTT is applied to convert the product to the coefficient embedding domain. When the polynomials are in RNS form, both the NTT and iNTT are decomposed to n concurrent subtasks. Finally, we compute the outer product result by merging the iNTT-converted polynomial with $\mathbf{\Psi^{-1}}$.

C. NTT optimizations

Due to the pervasive usage of NTT and iNTT in HE, prior researchers proposed optimized implementations for NTT on CPUs [19], CUDA-enabled GPUs [25]–[28] and FPGAs [29], [30]. On the CPU end, SIMD instructions enable a wider data processing width to accelerate a broad range of applications [31]–[35]. Leveraging this architectural feature, Intel HEXL provides a CPU implementation of the radix-2 negacyclic NTT using Intel AVX512 instructions [19] and Harvey's lazy modular reduction approach [36]. GPU-accelerated NTT

implementations typically adopt the hierarchical algorithm first presented by Microsoft Research for the Discrete Fourier Transform (DFT) [37]. In [27], researchers implemented the hierarchical NTT with twiddle factors, which are multiplicative constants in the butterfly computation stage (i.e. W in Algorithm 1), cached in shared memory. Rather than caching twiddle factors, in [28], Kim et al. computed some twiddle factors on-the-fly to reduce the cost of modular multiplication and the memory access number of NTT. In [26], Goey et al. considered the built-in warp shuffling mechanism of CUDA-enabled GPUs to optimize NTT.

The hierarchical NTT implementation computes the NTT in three or four phases [26], [37]. An N-point NTT sequence is first partitioned into two dimensions $N = N_\alpha \cdot N_\beta$ and then N_α NTT workloads are proceeded simultaneously, where each workload computes an N_β-point NTT. After this column-wise NTT phase is completed, all elements are multiplied by their corresponding twiddle factors and stored to the global memory. In the next phase, N_β simultaneous row-wise N_α-point NTTs are computed followed by a transpose before storing back to the global memory. N_α and N_β are selected to fit the size of shared memory on GPUs. Considering both the RNS representation of NTT and the batched processing opportunities in real-world applications can provide us with sufficient parallelisms, we adopt the staged NTT implementation rather than the hierarchical NTT implementation in this chapter.

D. An Overview of Intel GPUs

We use the Intel Gen11 GPU as an example [38] to elaborate the hierarchical architecture of Intel GPUs. An Intel GPU contains a set of execution units (EU), where each EU supports up to seven simultaneous hardware threads, namely EU threads. In each EU, there is a pair of 128-bit SIMD ALUs, which support both floating-point and integer computations. Each of these simultaneous hardware threads has a 4KB general register file (GRF). So, an EU contains $7 \times 4\text{KB} = 28\text{KB}$ GRF. Meanwhile, GRF can be viewed as a continuous storage area holding a vector of 16-bit or 32-bit elements. For most Intel Gen11 GPUs, 8 EUs are aggregated into 1 Subslice. EUs in each Subslice can share data and communicate with each other through a 64KB highly banked data structure — shared local memory (SLM). SLM is accessible to all EUs in a Subslice but is private to EUs outside of this Subslice. Not only supporting a shared storage unit, Subslices also possess their own thread dispatchers and instruction caches. Eight Subslices further group into a Slice, while additional logic such as geometry and L3 cache are integrated accordingly.

III. System Architecture

A. Overview

XeHE FHE GPU accelerating library has been designed to work as a backend of Microsoft SEAL, one of the most popular open-source HE software packages and HE APIs. The overall XeHE architecture is depicted on Fig. 1.

The XeHE architecture consists of two major parts: a separate Microsoft SEAL plugin built into the Microsoft SEAL

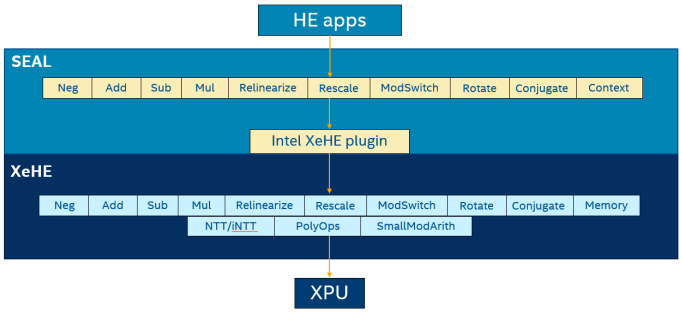


Fig. 1: SEAL XeHE integration

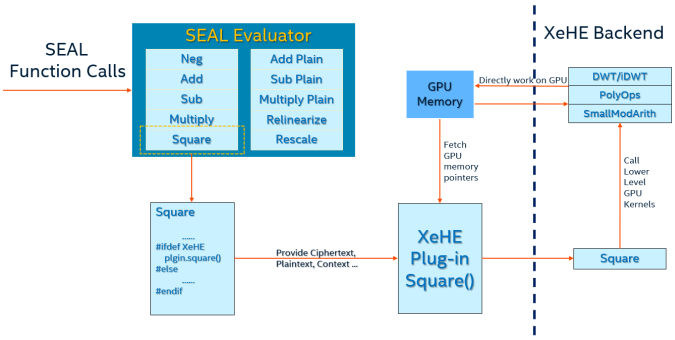


Fig. 2: SEAL-XeHE cooperation

binary and XeHE back-end library linked to Microsoft SEAL library dynamically. This design choice allows to reuse the XeHE back-end with other popular HE front-ends (Palisade, etc). Fig. 2 depicts the cooperation between SEAL, XeHE plugin and XeHE back-end, using SEAL Square() API as an concrete example. In particular, XeHE plugin converts SEAL data objects (Ciphertext and/or Plaintext) into XeHE Memory objects. This abstracts out GPU memory allocation, de-allocation and buffer's lifetime. Subsections III-E and III-G offer a more detailed discussion. The plugin routes calls XeHE backend APIs to execute HE operators on the GPU.

For any SEAL object participating in the GPU-bound operation, the **on-gpu** status is assigned to it, as shown in the subsection III-C. The deferred allocations and data transfers are executed right before XeHE plugin calls.

In addition, the XeHE library is responsible for executing HE operators on GPU. To support the XeHE-accelerated software design constraints (shown in III-B), we also maintain the *GPU memory cache* (Sec. III-E), manage the GPU *buffers lifetime* (Sec. III-G), control the asynchronicity (Sec. III-B2). The performance profiling events (Sec. III-I) are also supported to benchmark all of the instruction and algorithm level optimizations enabled in our HE library.

B. Design principles and constraints

During the development, we maintain the following design principals and constraints:

1) To keep the level of programmability as close to what Microsoft SEAL provides as possible: To that end we strive to keep the following features.

- an usage of APIs, C++ constructs, classes and objects exposed by Microsoft SEAL with minimum restriction.
- an easy adoption of the accelerated version of Microsoft SEAL.
- an integration with an upstream software stack and tools: HE applications, Microsoft SEAL HE compiler EVA, HE performance and verification tool chains (HEBench) etc.

2) Asynchronicity with the HOST: In the CKKS scheme, an input message is first encoded and then encrypted to generate cipher-texts, and XeHE computes directly on the already-encrypted messages, following the HE definition. The HOST software stack is responsible for encoding and encrypting the input data, applications control flow through the SEAL APIs, decrypting and decoding results at the end of computations.

The XeHE library is responsible for moving data to and from accelerator. The library also implements corresponding HE operators normally comprised of a sequence of GPU kernels. It works asynchronously from the HOST and frees the host to do other HE application-related tasks and house-keeping. Once a common security parameters-related data and Ciphertext operands are sent to the GPU, the synchronization with the HOST becomes unnecessary and the computation on the GPU starts as soon as the first kernel of the computational graph is submitted. Once all GPU computations are completed, XeHE synchronizes with the HOST and returns back to it the results of calculations. The results are then decrypted and decoded. See Fig. 5 depicting the asynchronous control flow of XeHE library and subsection III-F.

3) Scalability: FHE is extremely compute intensive, and it contains embarrassingly parallel workloads. Our software design allows an easy scale-up when deploying on a new accelerating device with a higher computation throughput and more compute unit or tiles. Each tile is recognized as a separate device and all tiles can work asynchronously and in parallel. Similarly with its transparent client-server architecture XeHE library allows easy scaling-out with other compute nodes.

4) Client-server control flow: The client-server architecture built into the XeHE design serves as a foundation for implementing scalability and asynchronicity with the HOST. The "HOST" client may run locally and "ACCELERATOR DEVICE" can process HE operators and the application logic locally or/and remotely in the cloud or on a separate dedicated server. Figure 3 depicts the client-server XeHE design.

C. *.gpu() method and its repercussions*

To conform with our "Programmability" design constraint, we have introduced **.gpu()** method for SEAL *Plaintext* and *Ciphertext* objects. This method triggers offloading for execution on GPU. The object still operates exactly the same way. If the operand was previously marked with **.gpu()**, then upon the SEAL Evaluator API call the operation will be executed on GPU. This transparent execution makes the porting to the accelerated version much more intuitive and debug-able.

Below can be found a code snippet from the HE multiply-add functionality test in XeHE test suit. The snippet is heavily commented to describe how exactly the implementation works under the hood.

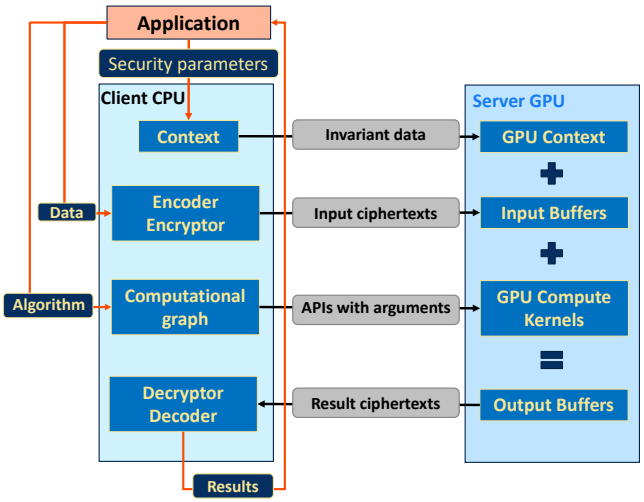


Fig. 3: Client (CPU)/Server (GPU) control/data flow.

```
.
// functionality verification snippet
//
// SEAL SW runs
// with XeHE plugin and XeHE backend

// evaluator, encryptor, decryptor, encoder
// are all instantiated with proper security
    parameters
// Ciphertext objects
// encrypted1, encrypted2, encrypted3
// have been properly encrypted
// Plaintext objects
// plainRes, xe_plainRes are properly instantiated

// output vector is the result from CPU-based
    calculations
// xe_output vector is the result from GPU-based
    calculations

vector<complex<double>> output(slot_size);
vector<complex<double>> xe_output(slot_size);

// copy constructors to save input data intact
// for the GPU verification
seal::Ciphertext cpu_encrypted1(encrypted1);
seal::Ciphertext cpu_encrypted2(encrypted2);
seal::Ciphertext cpu_encrypted3(encrypted3);


// CPU calculation
seal::Ciphertext cpu_encrypted_tmp;
// HE multiply on CPU
evaluator.multiply_inplace(cpu_encrypted1,
    cpu_encrypted2);
// HE add on CPU
evaluator.add(cpu_encrypted1, cpu_encrypted3,
    cpu_encrypted_tmp);

// decryption, decoding
// are always on CPU
decryptor.decrypt(cpu_encrypted_tmp, plainRes);
encoder.decode(plainRes, output);

// GPU calculation
// copy constructors
```

```
// encrypted data are still in system memory
seal::Ciphertext xe_encrypted1(encrypted1);
seal::Ciphertext xe_encrypted2(encrypted2);
seal::Ciphertext xe_encrypted3(encrypted3);

// C++ constructor, no data allocated
seal::Ciphertext xe_encrypted_tmp;

// the ciphertext object xe_encrypted1
// is touched with .gpu() call
// no data allocation or data movement have occurred
xe_encrypted1.gpu();

// HE multiply operator has been called.
// since xe_encrypted1 has been ".gpu() touched"
// the operation is going to be routed to GPU processing
// any other operand participating in GPU-bound HE operation involving xe_encrypted1
// is going to be ".gpu() touched" implicitly.

// following HE operators are going to be routed to GPU processing
// thanks to ".gpu() touched" xe_encrypted1 Ciphertext object

// GPU memory allocation and data upload occur
// only at the point of the XeHE library API invocation
// thanks to the differed ("lazy") approach (see subsection "Buffer lifetime control").

evaluator.multiply_inplace(xe_encrypted1, xe_encrypted2);

// Here xe_encrypted3, xe_encrypted_tmp are going to be ".gpu() touched" implicitly;
// xe_encrypted3 data is going to be uploaded to GPU memory;
// xe_encrypted1 is in GPU memory already.

evaluator.add(xe_encrypted1, xe_encrypted3, xe_encrypted_tmp);

// all previous GPU operation are executed asynchronously;
// synchronisation occurs only at this point

// decryption, decoding
// are always on the host

decryptor.decrypt(xe_encrypted_tmp, xe_plainRes);
encoder.decode(xe_plainRes, xe_output);

// next is to verify output and xe_output vectors are close enough
// ...
```

Listing 1: SEAL/XeHE code interleave

Below is a short snippet of the SEAL Evaluator class when it is compiled with XeHE plugin showing **.gpu()** effect on the computational flow.

```

//SEAL_USE_INTEL_XEHE flag controls both the build and compilation processes.
//multiply_inplace_ckks() is a XeHE plugin API and a peer to CPU ckks_multiply version.

void Evaluator::multiply_inplace(
    Ciphertext &encrypted1,
    const Ciphertext &encrypted2,
    MemoryPoolHandle pool)
    {
        auto context_data_ptr = context_.first_context_data();
        switch (context_data_ptr->parms().scheme())
        {
        case scheme_type::ckks:
#ifdef SEAL_USE_INTEL_XEHE
// GPU routed code
            if (encrypted1.on_gpu() || encrypted2.on_gpu())
            {
                get_xehe_plugin().multiply_inplace_ckks(
                    encrypted1,
                    encrypted2);
            }
            else
#endif
// Original SEAL code
            {
                ckks_multiply(encrypted1,
                    encrypted2,
                    pool);
            }
            break;
        default:
            throw invalid_argument(
            "unsupported scheme");
        }
    }
```

Listing 2: SEAL Evaluator XeHE plugin call

D. *Deferred ("lazy") allocation and data transfer*

Not all **.gpu()**-marked Ciphertext objects might reach the actual GPU calculation, due to, for example, going out of scope. To that end, the GPU memory allocation and data upload to the GPU memory are deferred and done *lazily*. The allocations and uploads happen only at the point of actual XeHE library API call. Exactly when Ciphertext objects host data has to appear in GPU memory to serve as inputs or outputs of the call. The following pseudo C++ code describes states and the state machine driving the *lazy* GPU memory allocation and data upload. Comments below should help the reader to get a sense of the state machine design and its complexity.

```

//inside SEAL::Cipheretxt class the following member variables controlling in "lazy" memory allocation and data upload.

class Ciphertext{
    bool m_gpu = false; //true - .gpu() touched; destine to GPU acceleration.
    bool m_dirty = false; // true - object's host buffer has been newly filled or updated.
    bool m_on_gpu = false; // true - the GPU data has been allocated and the host buffer has been uploaded to GPU.
    size_t m_size = 0; // the HOST buffer size has to be == GPU memory size.
    XeHEBuffer* m_gpu_buf = nullptr; // abstract representation of the XeHE GPU memory buffer
}

// on ciphertext encryption
seal:: Encryptor::encrypt(const Plaintext &pt, Ciphertext & ct)
```

```
{
     // the dirty flag is set to true
     // signalling the host memory buffer update.
     // no GPU memory allocation occurs.
     ct.m_dirty = true;
}

// on Ciphertext object copy constructor
seal::Ciphertext(const Ciphertext & other){
    // if the source object is touched the target is going to be touched implicitly as well.

    m_gpu = other.m_gpu;

    // Implicit allocation and data copy

    // if the source is already in GPU memory
    if (other.m_gpu && other.m_on_gpu)
    {
        // allocate target gpu memory
        m_gpu_buf = gpu_alloc(other.m_size);

        // copy GPU data from source to target
        copy(*this.m_gpu_buf, other.m_gpu_buf, m_size);

        m_on_gpu = true;
        m_size = other.size;
     }
     // other functionality
     ...
 }

// on execution of XeHE API call
plugin::XeHE_Evaluator::SomeApi(
    const Ciphertext & ct_in,
    Ciphertext & ct_out, ...){

    // assume both ct_in and ct_out are gpu bound (".gpu() touched") at this point of execution.

    // ct_in state machine
    bool new = false;

    // if there is no gpu memory, allocate it
    if ((new = !ct_in.m_on_gpu)) {
        ct_in.m_gpu_buff = new_buffer(m_size);
        ct_in.m_on_gpu = true;
    }

    // if this is a new buffer or the old one has been updated by the host,
    // upload data to GPU memory
    if (new || ct_in.m_dirty) {
        ct_in.upload(ct_in.m_size);
        ct_in.m_dirty = false;
    }

    // if the operation requires the ciphertext to result in a new size
    // allocate new buffer and copy old content into it
    new_size = get_new_size();
    if (ct_in.m_on_gpu && ct_in.m_size != new_size)
    {
        new_buf = new_buffer(new_size);
        copy(new_buf, ct_in.m_gpu_buf, ct_in.m_size);
        ct_in.m_gpu_buff = new_buf;
        ct_in.m_size = new_size;
    }

    // similar state machine is applied to ct_out object
    // taking into the consideration its output properties

    // the following routine executes an XeHE library Api
    // it operates only on the XeHE GPU memory buffer abstractions
    // not knowing anything about SEAL objects and classes
    xehe::XeHE.SomeApi(
        ct_in.m_gpu_buf,
        ct_out.m_gpu_buf,
        ...);
}
```

Listing 3: Deferred (*lazy*) memory allocation and upload state machine

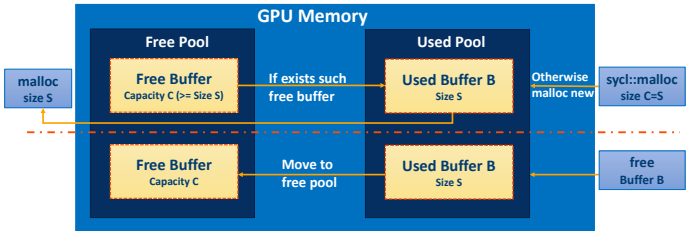


Fig. 4: Memory cache design

E. GPU Memory cache

To reduce the overhead introduced by runtime GPU memory allocation, a GPU memory cache mechanism has been built into XeHE library design, as shown in Figure 4. The motivation for a GPU memory cache is twofold. First, GPU devices usually have a noticeable latency between the request and action of memory allocation. Second, the required memory sizes for ciphertexts often lie in a small set of sizes related to static security parameters. Therefore, temporary memory buffers between nodes of the HE computational graph are likely to have similar sizes drastically simplifying the reuse policy of previously allocated memory buffers.

Similar to Microsoft SEAL, XeHE library introduces a memory pool to collect and re-assign outdated GPU memory buffer pointers in the HE pipeline. A request for a GPU memory buffer pointer is routed through the memory cache to check if there exists a memory buffer in the free pool with a capacity larger than the requested size. If such free memory buffer is found, pointer of the memory buffer is reassigned to the request and moved into allocated (or active) pool (list). Upon freeing a used memory buffer, the GPU memory pointer is moved back to the free pool for potential reuse but the memory buffer is kept allocated. Actual GPU memory allocations are only performed when the free pool has no suitable match. Therefore, most of the GPU memory buffer request will only involve pointer assignments, drastically reducing memory buffer allocation latency compared to real GPU memory allocations. API for the *memcache* should also include a method to flush the pool of free memory buffers. Shown below is an example of simple *memcache* class to achieve this:

```
typedef struct {
    void* dev_ptr_;
    size_t capacity_;
} FREE_STRUCT;

typedef struct {
    size_t capacity_;
    // this union structure will be discussed later
    union {
        uint64_t retained_ : 1;
        uint64_t ready_to_free : 1;
    };
} ALLOC_STRUCT;

class Memcache{
public:
    Memcache(const cl::sycl::queue & q):q_(q){};

    // deallocate all cached buffers - EOL memcache
    ~Memcache(void);

    // deallocation of all free but unused buffers
    void free_pool_free_buffers(void);

    // deallocate free buffers and deactivate cache, so that all follow up free requests are real deallocations
    void dealloc(void);

    // API to free an allocated buffer
    void pool_free(void * data, size_t capacity, bool uncached = false);

    // API to allocate a buffer of type T
    template<typename T>
    T * pool_alloc(size_t buffer_size, size_t & capacity, bool uncached = false);

private:
    // utility function to add a pointer&capacity to allocated pool
    void add_to_alloc_pool(void* new_ptr, size_t capacity, uint32_t retained = 0);

    // utility function to remove a pointer from allocated pool
    void remove_from_alloc_pool(void* ptr);

    // utility function to add a pointer&capacity to free pool
    void add_to_free_pool(void* ptr, size_t capacity, bool uncached = false);

    // utility function to find a pointer with capacity>size and remove from free pool
    void* remove_from_free_pool(size_t size, size_t& capacity);

    // utility function to check if a pointer is retained
    uint32_t is_retained(void* ptr);

    // utility function to check if a pointer is ready to free
    uint32_t is_ready_to_free(void* ptr);

    // utility function to get the free pool instance for the memcahce
    std::vector<FREE_STRUCT>& get_free_pool(void);

    // utility function to get the allocated pool instance for the memcahce
    std::map<void*, ALLOC_STRUCT>& get_alloc_pool(void);

    // utility function to get the corresponding queue instance for the memcahce
    cl::sycl::queue & get_queue();

    // define the state of the cache
    // if at least 1 buffer is in the alloc cache, then it's active
    // dealloc call switches it back to inactive
    // While inactive, any buffer freeing leads to normal deallocation, flushing the cache.
    bool mem_cache_active_ = true;

    std::vector<FREE_STRUCT> free_pool_;
    std::map<void*, ALLOC_STRUCT> alloc_pool_;

    const cl::sycl::queue& q_;

    // separate locks for free&alloc pool
    // to ensure thread safe manipulations on pools
    std::mutex free_lock;
    std::mutex alloc_lock;

}; // class memcache
```

Listing 4: Memory cache class and methods

Separately, for illustration purposes we have listed the implementation of the memcache allocation API method in *memcache* class:

```
template<typename T>
T * pool_alloc(size_t buffer_size, size_t & capacity, bool uncached = false)
{
    T *ret = nullptr;
    if (buffer_size <= 0)
        std::cout << "Warning: tried to alloc 0 size buffer" << std::endl;

    capacity = buffer_size;
    size_t new_capacity = capacity;

    // if the cache is deactivated
    if (uncached || !pool_memcache_active()){
        // do allocation without involving memcache
        ret = cl::sycl::malloc_device<T>(
                buffer_size, get_queue());
    }
    else
    {
        // otherwise use the memcache mechanism
        bool new_malloc = true;

        // thread safety guard
        {
            // acquire free&alloc locks
            std::scoped_lock lock(
                 free_lock, alloc_lock);

            // searching for the free buffer
            // with size >= requested capacity
            auto pooled_free_ptr
            = remove_from_free_pool(
                  buffer_size, new_capacity);

            capacity = new_capacity;

            // if there exist such in the free pool
            if (pooled_free_ptr != nullptr){
                ret = (T*)pooled_free_ptr;
                // add it into the alloc pool
                add_to_alloc_pool(
                     (void*)ret, capacity);
```

```
                new_malloc = false;
            }
        // both locks will be released out of scope
        }

        // new allocation needed
        if (new_malloc){
            // call GPU memory allocator
            ret = cl::sycl::malloc_device<T>(
                 buffer_size, get_queue());
            // acquire only alloc lock
            std::lock_guard<std::mutex> lk(
                alloc_lock);
            // insert into the alloc pool
            add_to_alloc_pool((void*)ret, capacity);
        // lock will be released out of scope
        }
    }

    return ret;
}
```

Listing 5: Pool allocation method in *memcache* class

Similarly, the implementation of API method for marking allocated pointer as available in the pool is shown here:

```
void pool_free(void * data, size_t capacity,
     bool uncached = false)
{
    if (!is_retained(data))
    {
        // thread safety guard
        // acquire free&alloc locks
        std::scoped_lock lock(free_lock,alloc_lock);

        // remove the pointer from alloc pool
        remove_from_alloc_pool(data);

        // add the pointer back to free pool
        add_to_free_pool(data, capacity, uncached);
        // locks will be released out of scope
    }
}
```

Listing 6: Pool free method in memcache class

F. Asynchronous execution

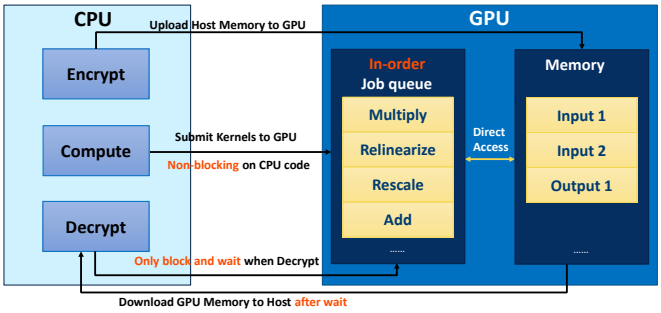


Fig. 5: Asynchronous execution scheme

To achieve the CPU-GPU asynchronous execution we relied on the following design constraints and principles.

1) XeHE plugin is instantiated together with each new instance of **SEAL::Evaluator class**, Thus all global XeHE parameters are separated from one instance to another.
2) XeHE dynamic library APIs are thread-safe by design including all basic operators and memory management (memory cache) allowing to execute any API in different threads and processes concurrently.
3) We assume a single in-order DPC++ queue is assigned to a single computational graph (SEAL HE application). Subsection III-H discusses a multi-queue implementation in the multi-tile device environment. Similarly, an application should explicitly assign a particular queue to a particular HE computational graph in order to run multiple computational graphs concurrently on the same device tile.
4) We have a *Buffer lifetime control* system implemented for GPU buffers, so this lets GPU-bound algorithms and HE operators to run asynchronously from the HOST. This happens until an application specific synchronization point, for example, returning results of a HE calculation to the HOST.

Here we focus on capabilities XeHE exposes to an SEAL application for controlling asynchronous computation and to manage synchronization points. The section also describes internal implementation details of the synchronization protocol.

In the simplest scenario the SEAL application might not even be aware that it runs asynchronously with GPU. If any Ciphertext object has been marked with **.gpu()** then operations involving those objects are routed for the GPU execution. In that case all burden of dispatching asynchronous kernels is on the cooperation of XeHE plugin and XeHE library. In this most common scenario, the method **SEAL::Decryptor.decrypt()** becomes the implicit synchronization point.

Inside the API, the plugin calls 2 XeHE methods: **EndComputeFrame()** (see III-D) and **Ciphertext.download()**. The first method forces the GPU to synchronize with the host and guarantees completion of any operations on GPU data at this point in the computational graph. The second method moves the GPU data into a particular Ciphertext host-side buffer to be processed by HOST later. Usually processing involves the decryption and decoding, in order to obtain a plain text result of the HE computation.

The Fig.5 depicts the scenario and a SEAL-based application works as is. The first call to **SEAL::Evaluator** API forces static data depended on security parameters to be uploaded to GPU. After that all SEAL HE operators (GPU bound) are launched in the same order as in the SEAL computational graph, and asynchronously, until the graph reaches **SEAL::Decryptor.decrypt()** API where synchronization and download occur.

In the previous scenario the synchronization being exercised by **download()** method is indeed a NOP operation since it is already been done inside **EndComputeFrame()** API. However the **download()** API can work independently. This is very handy in debugging mode, allowing to inspect Ciphertext GPU data at any point of computational graph. However it does not change any buffer's lifetime states as **EndComputeFrame()** handles the *buffer lifetime control* (III-G). The third scenario involves using the **Start/EndComputeFrame()** method pair as

described in the subsection III-G. In general, all 3 scenarios can be applied to the same computation graph at different points of the control flow.

For debugging purpose, XeHE also implements a fine grain synchronization based on the *wait* mechanism. This mechanism provides an easy way to compare the behavior of the same kernel under synchronous and asynchronous execution. XeHE runs each computational graph on a separate in-order queue, which makes the synchronization mechanism rather simple. We may rely on DPC++ **wait()** call instead of on more sophisticated event constructs. Please see the DPC++ pseudo-code snippet for a better understanding of the concept. The *wait* mechanism together with the *simulation* infrastructure (see a later section) have been used extensively to achieve a fast CPU-GPU functional correctness convergence.

```
1
2  // XeHE_relinearize API is implemented as a sequence
       of kernels.
3  // The API's last parameter is a Boolean wait;
4  // each kernel in the underlying implementation
       series have a Boolean wait as well;
5  // if XeHE_relinearize  wait is set to false,
6  // than the sequence runs in-order, asynchronously
       to the HOST;
7  // if it's set to true, the sequence is synchronized
       with the HOST after the last kernel completes
       its execution
8  // allowing for the inspection of the output.
9  // Since every kernel in the sequence has its own
       Boolean wait
10 // parameter it can be turned on at any time
11 // to inspect and to verify (after recompilation)
       the inputs and outputs of the kernel.
12
13 void XeHE_relinearize(sycl::queue q,
14     ...,
15     bool wait
16 )
17 {
18    inverse_ntt_negacyclic_harvey(
19         ...,
20         false  // fine grain synchronization
21         );
22     rln_keys_mod(
23         ...,
24         false // fine grain synchronization
25         );
26
27     ...
28
29     rln_ctmodqi_ctmodqk_modqi(
30         ...,
31         false // fine grain synchronization
32         );
33
34     // synchronize the entire sequence
35     if (wait)
36     {
37         q.wait();
38     }
39 }
```

Listing 7: XeHE wait tree

G. Buffer lifetime control

Buffer lifetime control is needed to support the asynchronous execution model in the coordination with *GPU Memory cache*. The HOST-GPU asynchronous interaction means that a Ciphertext object might request de-allocation of the underlining GPU buffer before the completion of an asynchronous GPU operation, for example when Ciphertext is going out of C++ scope. To avoid confusion among ciphertext, buffer and memcache, Figure 6 illustrates the relationship among these three classes in our software architecture.

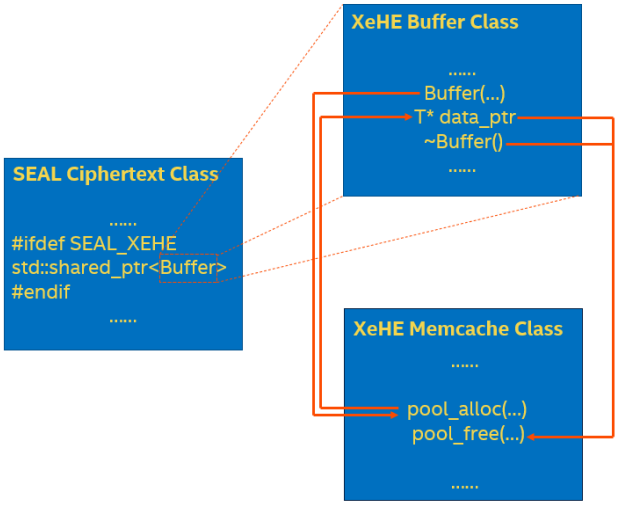


Fig. 6: Relationship among ciphertext, buffer and memcache

The buffer lifetime control alleviates this problem and introduces a new pair of XeHE methods.

StartComputeFrame() sets a globally visible flag **"retain_buffers"** to true and more (see below)

EndComputeFrame() sets **"retain_buffers"** to false and more (see below).

A Ciphertext object keeps shared_ptr to an underlying GPU memory buffer object that abstracts GPU memory operation from the higher level stack. If the global *retain_buffers* bit is **ON**, then the memory cache allocation routine sets the *retain* bit **ON** for every newly allocated GPU buffer that is put into the allocated (**active**) buffers list as described in Sec. III-E.

At any point of computation when a Ciphertext object is destructed, the underlying GPU memory buffer shared_ptr counter goes down. When it reaches 0, then the request for the GPU memory buffer deallocation is passed to the memory cache subsystem. If the buffer has "retain" bit set, then the memory cache, instead of moving the GPU buffer pointer to the free "non-active" list, keeps it in "active" list and sets "ready_to_delete", bit on.

StartComputeFrame() API synchronizes with host, sets "retain_buffers" to true and runs a loop over all entries in allocated ("active") list and sets the "retain" bit to true for each of them.

EndComputeFrame() API synchronizes with the host, switches the global "retain_buffer" bit off and runs a loop over the memory cache active buffers setting retain bits off and moving those with "ready_to_free" bit set into the free buffer list.

At the start of calculations **StartComputeFrame()** is called implicitly.

On decrypting of a Ciphertext object, **EndComputeFrame()** is invoked implicitly.

This design permits SEAL API users not to worry about the lifetime of the underlying GPU buffers. Drawback of this approach is any GPU buffer allocated between the pair of **Start/EndComputeFrame** calls stays active and consumes scarce GPU off-chip memory. The simple remedy is to use the pair explicitly in the application control flow periodically – in this case it works as an explicit garbage collector.

H. Multi-tile scaling

Intel packages multiple computing tiles on a single board for scalable performance [39]. At the time of writing DPC++ runtime did not implicitly support the multi-tile submission at full performance, but expected to work in the future. That implies the memory independent workloads will not be distributed over all tiles of a multi-tile Intel GPU automatically. In order to maximize the utilization of multi-tile devices, XeHE maintains one queue for each tile and submit workloads to different queues. Fig. 8 shows the implementation details of the library's multi-queue DPC++ context: it checks whether multi-tile is supported on current device via DPC++/sycl partition functions, creates in-order queues for each (sub-)device (tile), and attaches the queues to the corresponding (sub-)device.

XeHE library achieves explicit multi-tile scaling by submitting workloads to the multiple queues, utilizing all the sub-devices initialized at DPC++ context creation. Workloads on different queues are assumed to be memory independent. The assumption is achievable by submitting independent HE computation graphs to different queues. That reflects real world applications where different clients always send independent computation requests. The assumption simplifies the memory management across multiple-tile device and supports a separate memory cache for each queue as mentioned in the section above. Also, exploiting the advantage of fast tile-to-tile shared memory, we can load the shared data, such as security parameter context, only on a specific tile at initialization and share it across the tiles at run-time. This will reduce initialization overhead and simplify the code structure without losing run-time performance.

```
class Context {
    bool igpu = true;
    std::vector<cl::sycl::queue> _queues;
    void generate_queue(bool select_gpu = true){
        if (select_gpu) {
            sycl::device RootDevice = sycl::device(
                intel_gpu_selector());
            std::vector<sycl::device> SubDevices;
            try {
            // check if sub devices (tile split) is supported on GPU device
                SubDevices = RootDevice.create_sub_devices
                <sycl::info::partition_property::partition_by_affinity_domain>
                (sycl::info::partition_affinity_domain::next_partitionable);
            }
            catch (...) {
                std::cout << "Sub devices are not supported\n";
                // only use the root device
                SubDevices.push_back(RootDevice);
            }
            // create in-order queues and attach to sub-devices
            sycl::context C(SubDevices);
            for (auto &D : SubDevices) {
                sycl::queue q;
                q = sycl::queue(C, D, sycl::property::queue::in_order());
                _queues.push_back(q);
            }
        } else {
            // create queue based on CPU device
            ...
        }
    }

public:
    Context(bool select_gpu = true){
        generate_queue(select_gpu);
        =}
    ...
};
```

Listing 8: DPC++ Context with multiple queue

I. DPC++ profiler and event class

To evaluate XeHE kernels performance during the optimization phase, the library includes a light-weight profiler. The profiler provides kernels timing statistics at run-time. The profiling session involves two steps: event collection and data processing.

In DPC++, each kernel launch is defined as an event that can be queried for profiling information such as submission timestamp, start timestamp, end timestamp, etc. At the first, the EventCollector object collects all the events happened at runtime and stores them in a map associated with the kernel's name, as shown in Fig. 9. If kernel is launched multiple times, then it will have multiple events recorded with EventCollector. The design of EventCollect ensures that all the necessary profiling information is gathered through the runtime, and they are ready to be processed by the DataProcessor object to produce useful profiling data.

The DataProcessor exposes to a user two sets of APIs for various purposes of analysis, as shown in Listing 10. The first set is kernel-level data display APIs, mainly targeted a single kernel performance data. These APIs group the kernel profiling data by the kernel name, processes the average GPU execution time per each kernel and displays the data directly to the terminal output. Another set is the high-level operation and file dump APIs. It gives a user ability to group the collected kernel profiles into a higher-level operation. It will also provide the proportion of each kernel's GPU time, as well as the NTT-related kernel proportion, given the external execution time of the high-level operation. This will help users to have a better understanding of their application-level operations from the kernel composition perspective. The API dumps the generated

operation table into files, providing user a simple and clear view of the profiling data.

```
class EventCollector{
public:
    // add event with the kernel name to record profilling info; this call is library-internal and not visible to the high-level application
    static void add_event(std::string name, cl::sycl::event e){
        if (!activated_) return;
        if (events_.find(name)==events_.end()){
            events_[name]={e};
        }
        else{
            events_[name].push_back(e);
        }
    }

private:
    inline static std::unordered_map<std::string, std::vector<cl::sycl::event>> events_;
    inline static bool activated_ = true;
}
```

Listing 9: Code snippet of event profiler

```
// Kernel-level data display APIs
// Interface for summarizing the average GPU time per kernel in the EventCollector
void process_events();

// Interfaces for clearing all the events collected by the EventCollector
void clear_events();

// High-level operation & file dump APIs
// Interface for grouping and processing all the kernels into a high-level operation per user's specification, append it to export table
void add_operation(std::string op_name, double avg_external_time, int loops);

// Interface for dumping the export table to a file
void export_table(std::string file_name);

// Interfaces for clearing the export table
void clear_export_table();
```

Listing 10: Public APIs of profiling data processing

IV. Other development strategies

A. Reusing trusted C++ routines

DPC++ design is targeting a plethora of computational devices with a single portable code. Its GPU-bound dialect appears syntactically, semantically and computationally (including arithmetic for int, int64, float and double) identical to a modern C++ targeting CPU.

A big boost in SW productivity might be gained with reusing trusted CPU subroutines. It is especially important in the privacy-preserving computing field. Most of HE operations is a deep "matreshka doll" of subroutines relying on correct implementation of underlying layer. This SW organization may also direct optimization efforts. Since the lowest level routine performance directly affect the entire stack (see V).

In particular, many HE operators rely on the NTT/iNTT algorithm that extensively utilizes the modulo multiplication that uses Barrett reduction and all of the above are based on integer multiplication and addition. Having an extensively tested and optimized version of the integer multiplication routine on CPU, we can invoke it without any modification for the GPU bound source and guarantee correctness and efficiency.

Utilizing templates of modern C++ language the XeHE can be easily adapted to any integer arithmetic width. The current version supports 32/64 bit arithmetic and nothing prevents it to go to wider types.

The "native" kernels in the project contains modulo and low-level arithmetic routines. All of them can be called interchangeably from HOST or GPU-bound kernels. Next section describes how we take advantage of these properties to achieve a faster HOST/GPU functionality convergence and better debugging capabilities. .

B. Simulation using overloads

We will illustrate the idea of reusing trusted C++ routines by the following code snippet. At the bottom of the snippet, reader may see two similar invocations with the same names and almost the same signatures. The only difference between signatures is a missing sycl::queue q parameter in one of them. All other parameters are syntactically and semantically the same. The routine without the **queue** parameter is a CPU bound, the routine with **queue** launches a GPU kernel. Both CPU and GPU bound routines call the same core kernel. The later calls the native integer modulo arithmetic operator. The operator can be called interchangeably either by CPU or by GPU routines as being discussed in previous subsection.

```
// XeHE core native arithmetic
#include "xe_uintarith_core.hpp"


// CPU/GPU common code

template<typename T>
void
kernel_coeff_prod_mod(
    int coeff_idx, int rns_idx, int poly_idx,
    int n, int q_base_size, int n_polys,
    const T* oprnd1, const T* oprnd2,
    const T* modulus, const T* inv_mod2, T* result)
    {

    uint64_t poly_offset = uint64_t(rns_idx) * n
            + uint64_t(poly_idx) * n * q_base_size;

    uint64_t global_idx = uint64_t(coeff_idx) +
    poly_offset;

    auto ret = xehe::native::mul_mod<T>(
        oprnd1[global_idx], oprnd2[global_idx],
        modulus[rns_idx], &inv_mod2[rns_idx * 2]);

    result[global_idx] = ret;
}

// CPU simulation
template<typename T>
void poly_coeff_prod_mod(int n_polys, int
    q_base_size, int n,
    const T* oprnd1, const T* oprnd2,
    const T* modulus, const T* inv_mod2,
```

```
    T* result) {

    // GPU work item loops simulation
    for (int p = 0; p < n_polys; ++p)
    {
       for (auto idx = 0; idx < n; ++idx) {
            for (auto rns_idx = 0;
                rns_idx < q_base_size;
                ++rns_idx) {

                xehe::kernels::
                kernel_coeff_prod_mod<T>(
                    idx, rns_idx, p,
                    n, q_base_size, n_polys,
                    oprnd1, oprnd2, modulus,
                    inv_mod2,
                    result);

          }
        }
    }

}

// GPU kernel launch
template<typename T>
class krnl_coeff_prod_mod;

template<typename T>
void poly_coeff_prod_mod(cl::sycl::queue& queue,
    int n_polys, int q_base_size, int n,
    const T* oprnd1, const T* oprnd2,
    const T* modulus, const T* inv_mod2,
    T* result, bool wait = false) {

    auto grid_range = sycl::range<3>(n_polys,
    q_base_size, n);

    auto e = queue.submit([&](cl::sycl::handler& h)
    {
        h.parallel_for<class krnl_coeff_prod_mod<T>>
        ({ grid_range }, [=](auto it)
        {
            int idx = it.get_id(2);
            int rns_idx = it.get_id(1);
            int poly_idx = it.get_id(0);

            xehe::kernels::kernel_coeff_prod_mod<T>(
            idx, rns_idx, poly_idx,
            n, q_base_size, n_polys,
            oprnd1, oprnd2, modulus,
            inv_mod2,
            result);

        });
    });

// Simulation and native GPU execution samples
{
// part of the testing suit
  sycl::queue q;


  // poly, poly2, xe_modulus, hd_poly_res are
    vectors of floats in system memory
  // keeping information identical to
  // poly_buf, poly2_buf, xe_mod_buf, poly_res_buf
    that are GPU bound memory buffers

  // other params
    ...

    xehe::native::poly_coeff_add_mod(
       n_polys, q_base_sz, n,
       poly, poly2, xe_modulus,
       hd_poly_res.data());

    xehe::native::poly_coeff_add_mod<T>(queue,
        n_polys, q_base_sz, n,
        poly_buf, poly2_buf,
        xe_mod_buf, poly_res_buf);

}
```

Listing 11: Simulation using overloads

By design, the same DPC++ code running on CPU or GPU should produce identical or close enough results. It requires however that the entire computational graph runs exclusively on CPU or GPU device. The advantage of our approach is an ability to freely mix GPU and CPU bound code. The approach is most effective in developing and debugging complex algorithms, involving long sequence of kernels, in step-by-step fashion. That is particularly true for complex HE operators. Our experience shows that it significantly speeds up the functionality convergence between CPU and GPU versions of complicated algorithms.

Attentive reader may notice that the presented naive simulation lacks more advanced features of GPU programming model, the local shared memory and an inter work-item communication among others. A more sophisticated simulation is very well possible and has been developed by one of the authors and it might be a subject for another publication.

V. Instruction and kernel level optimization techniques

In the following contents of this section, we present optimizations of our library from 2 different angles: instruction level, algorithm level.

A. Instruction-level Optimizations

Our HE library supports basic instructions such as addition, subtraction, multiplication and modular reduction – all are 64-bit integer (int64) operations. We explicitly select int64 because our goal has been to provide accelerated SEAL APIs on Intel GPUs transparently. This is the key reason why our current top-level software does not exactly fit to drive 32-bit integer (int32) calculations, although we envision to support both int32 and int64 eventually. Among these operations, the most expensive are modulus-related operations such as modular addition and modular multiplication. Although we can accelerate modular reduction using the Barrett reduction algorithm, which transforms the division operation to the less expensive multiplication operation, modular computations remain costly since no modern GPUs support int64 multiplication natively. Such multiplications are emulated at software level with the compiler support.

Based on these observations, we propose instruction-level optimizations from two aspects: 1) fusing modular multiplication with modular addition to reduce the number of modulo operations and 2) optimizing modular addition/multiplication from assembly level to remedy the compiler deficiency.

1) Fused modular multiplication-addition operation (`mad_mod`*):* Rather than eagerly applying modulo operation after both multiplication and addition, we propose to perform only one modulo operation after a *pair* of consecutive multiplication and addition operations, namely a `mad_mod` operation. We store the output of int64 multiplication in an 128-bit array. The potential overflow issue introduced by cancelling a modulus after addition is not a concern when both operands of addition are integers strictly less than 64 bits. This assumption holds because to assure a faster NTT transform, we adopt David Harvey's optimizations [36] following SEAL. Therefore, all of our ciphertexts are in the ring space under a integer modulus less than 60 bits.

```
#define ADDMOD_64_STR(y) "{\n" \
  ".decl temp0 v_type=G type=uq num_elts=" STR(y) "
    align=qword alias=<%0, 0>\n" \
  ".decl temp1 v_type=G type=uq num_elts=1 align=
    qword alias=<%3, 0>\n" \
  ".decl temp2 v_type=G type=q num_elts=" STR(y) "
    align=qword\n" \
  ".decl P1 v_type=P num_elts=" STR(y) "\n"\
  "add (M1, " STR(y) ") %0(0,0)<1> %1(0,0)<1;1,0>
    %2(0,0)<1;1,0>\n" \
  "cmp.lt (M1, " STR(y) ") P1 temp0(0,0)<1;1,0>
    temp1(0,0)<0;1,0>\n" \
  "(P1) sel (M1, " STR(y) ") temp2(0,0)<1> 0x0:d
    %3(0,0)<0;1,0>\n"\
  "add (M1, " STR(y) ") %0(0,0)<1> %0(0,0)<1;1,0>
    (-)temp2(0,0)<1;1,0>\n" \
  "}\n"
```

Listing 12: Intel vISA code snippet of int64 addmod

We review the assembly codes generated by the Intel DPC++ compiler and seek low-level optimization opportunities for the HE pipeline. We locate such opportunities in two of our core arithmetic operations: Unsigned Modular Addition (`add_mod`), and Unsigned Integer Multiplication (`mul64`). Once locating the underperforming compiler-generated codes, we resort to using inline assemble featuring Intel virtual-ISA (vISA) [40] to optimize these performance-sensitive parts of the program. The source code snippets are presented in Listing 12 and Listing 13. To better help a reader to understand the content, we present and refer to the pseudo codes when elaborating our optimization strategies.

1: add dst, src1, src2 2: cmp.lt P1, dst, modulus 3: (P1) sel modulus, 0x0, modulus 4: add dst, dst, (-)modulus	1: add dst, src1, src2 2: cmp.lt P1, dst, modulus 3: (P1) add dst, dst, (-)modulus
(a) Compiler-generated assembly	**(b) Hand-crafted assembly**

Fig. 7: Pseudo int64 addmod assembly

a) Unsigned Modular Addition (`add_mod`*):* Fig. 7(a) presents the compiler-generated sequence of `add_mod`. Two source operands (src1, src2), and the result is stored to the register (dst). If the summation exceeds the value of modulus, the result is added by the negative modulus; otherwise, no update is needed. The compiler suboptimally implements this logic by conditionally initializing the addend (modulus) and then updating the result. At line4 in Fig. 7(b), we directly perform a conditional addition by leveraging the optional guard predicate (P1) of `add` on Intel GPUs. Here we eliminate one instruction at the assembly level for this core HE arithmetic operation, which enables direct benefits to the whole HE pipeline.

1: mul temp, src2, src1 2: mulh temp1, src2, src1 3: mul temp2, src2, src1 4: add temp1, temp1, temp2 5: mul temp2, src2, src1 6: add temp1, temp1, temp2 7: mov dst_low, temp 8: mov dst_high, temp1	1: mul_low_high dst_low_high, src1, src2
(a) Compiler-generated assembly	**(b) Hand-crafted assembly**

Fig. 8: Pseudo mul64 assembly

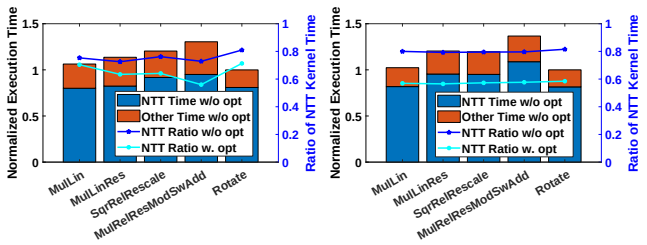
b) Unsigned Integer Multiplication (`mul64`*):* Another example where our hand-crafted assembly code outperforms the compiler-generated instruction sequence can be found in int64 multiplication. Fig. 8(a) shows the compiler-generated instruction sequence to multiply two 64-bit integers, producing an 128-bit result which is stored in two 64-bit registers (dst_high, dst_low). The instruction `mul` takes two 64bit operands to compute the lower 64 bits of the multiplication result, while `mulh` computes the higher 64 bits.

Although the compiler-generated code provides us with a correct result (the lower 64 bits of int64 multiplication), it also computes the higher 64 bits of int64 multiplication, which are redundant in our case. In order to address this issue, we adopt the built-in `mul_low_high` operator to explicitly compute the lower 64-bit multiplication result, as shown in Fig. 8(b). To elaborate, `mul_low_high` receives two int32 operands (cast from int64) and stores both the lower and higher 32 bits of the result in a 64-bit destination [41].

```
#define MUL_UINT_OPT_64_STR(y) "{\n" \
  ".decl temp1%= v_type=G type=ud num_elts=" STR(y)
    " align=hword\n" \
  ".decl temp2%= v_type=G type=ud num_elts=" STR(y)
    " align=hword\n" \
  "mov (M1, " STR(y) ") temp1%=(0,0)<1> %1(0,0)
    <1;1,0>\n" \
  "mov (M1, " STR(y) ") temp2%=(0,0)<1> %2(0,0)
    <1;1,0>\n" \
  "mul (M1, " STR(y) ") %0(0,0)<1> temp1%=(0,0)
    <1;1,0> temp2%=(0,0)<1;1,0>\n" \
  "}\n"
```

Listing 13: Intel vISA code snippet of mul64

At the time of writing, this presents an example of a compilation deficiency related to variables' type-casting. By default, the compiler minimizes the number of type-casting instructions, but it is overall detrimental in the above case. An integer multiplication, where both operands are int32, is more efficient than a longer emulated implementation whose both operands are of type int64. Our inline assembly bypasses this deficiency, yielding a significant reduction in instruction count from our original int64 multiplication implementation. As will be shown in Section VI, optimizations aimed at our core arithmetic operations greatly impact the performance of HE.



(a) Profiling on Device1 (b) Profiling on Device2

Fig. 9: Profiling for HE routines

B. Algorithmic level optimizations (NTT)

An efficient NTT implementation is crucial for HE computations since it accounts for a substantial percentage of the total HE computation time [28], [42], [43]. Figure 9 presents the relative execution time of five HE evaluation routines and the percentage of NTT in each routine before and after optimizing NTT kernels on two latest Intel GPUs, Device1 and Device2. We observe that NTT accounts for 79.99% and 75.64% of the total execution time in average on these two platforms. After applying optimizations as shown in Fig. 17 and 19, these NTT kernel ratios remain greater than 56% on both devices.

1) Naive radix-2 NTT: We start NTT optimizations from the most naive radix-2 implementation. Listing 14 presents the SYCL code snippet of the naive NTT implementation. A three-dimensional SYCL kernel corresponding to the three layers of the for loop is invoked when deploying this algorithm on Intel GPUs. This SYCL kernel, which needs to load NTT elements from the global memory, is launched at each round of the NTT computation. It is worth mentioning that in Listing 14 the SYCL kernel is launched through a lambda expression instance of a function object. This is a feature elaborated in a SYCL handbook [44].

This reference implementation of NTT distributes rounds of radix-2 NTT butterfly operations among work-items, which are analogs to CUDA threads. In each round of the NTT computation, all the work-items compute their own butterfly operations and exchange data with other work-items using the global memory. More specifically, the k-th element will exchange with the $k + gap$ th element, while the exchanging gap sizes are halved after each round of NTT until it becomes equal to 1. Accordingly, an N-point NTT is executed $\log(N)$ rounds throughout the computation. For each round of the NTT butterfly computation, one accesses the global memory $2N$ times. Here we multiply it by two because of both load and store operations. We ignore the twiddle factor memory access number in this semi-quantitative analysis.

```
template <typename T, int log_Unroll>
class RnsDwtGap {
public:
  /* omitting the constructor details */
  RnsDwtGap(/*input of the constructor*/) {/*...*/}
  void kernel(uint64_t poly_idx, uint64_t q,
    uint64_t ind) const {
    /* omitting codes to compute address offsets */
    auto x = values_ + global_offset + offset +
    poly_offset;
    auto y = x + gap_;
    for (int k = 0; k < (1<< log_Unroll); k++) {
      /* butterfly algorithm */
      auto u = dwt_guard(*x, two_times_modulus);
      auto v = dwt_mul_root(*y, r_op, r_quo, modulus
    );
      *x++ = dwt_add(u, v);
      *y++ = dwt_sub(u, v, two_times_modulus);
    }
  }

  void operator()[[intel::reqd_sub_group_size(
    SUB_GROUP_SIZE)]](cl::sycl::id<3> ind) const{
    uint64_t i = ind[2];
    uint64_t q = ind[1];
    uint64_t poly_idx = ind[0];
    kernel(poly_idx, q, i);
  }

}

/* naive radix-2 NTT */
for (i = 0; i < poly_num_; ++i) {
  for (q = 0; q < q_base_size_; ++q) {
    for (j = 0; j < (1 << (log_m + log_gap)); ++j) {
      /* launch SYCL kernel at every round of NTT */
      RnsDwtGap<T, 0>(/*input of the constructor*/).
    kernel(i, q, j);
    }
  }
}
```

Listing 14: SYCL code snippet of the naive NTT

Inside of the SYCL kernel, each work-item computes a two-point butterfly computation, which is accelerated using Algorithm 1 [36]. Since the output X', Y' of Algorithm 1 are both in $[0, 4p)$, to ensure all elements of the output NTT sequence falls inside of the interval $[0, p)$, a last round offsetting needs to be appended to the end of NTT computations. Therefore, the naive implementation of an N-point NTT needs to access the global memory $2N \log(N)$ times for the NTT and $2N$ extra times for last round processing. This kernel reaches only 10.08% of the peak performance for a 32K-point, 1024-instance NTT as shown in Fig. 13(b). Here the number of instances refers to the number of polynomials in Fig. 12. More specifically, 1024-instance NTT denotes 1024 batched instances of N-point NTT computation.

Algorithm 1: Optimized NTT butterfly computation

Input: $0 \leq X, Y \leq 4p$, $p < \beta/4$,
$0 < W < p, 0 < \lfloor W\beta/p \rfloor = W' < \beta$;
Output:
$X' = X + WY \mod p$;
$Y' = X - WY \mod p$;
$0 \leq X', Y' \leq 4p$;
if $X \geqslant 2p$ **then** $X \leftarrow X - 2p$;
$Q \leftarrow \lfloor W'Y/\beta \rfloor$;
$T \leftarrow (WY - Qp) \mod \beta$;
$X' \leftarrow X + T$;
$Y' \leftarrow X - T + 2p$;
return X', Y'

```
gap = N; m = 1; rounds = 1;
/* exchanging data via global memory */
for (; gap > TER_SLM_GAP_SZ; rounds += m, m *= 2;
    gap /= 2){
```

```
  auto grid_range = sycl::range<3>(poly_num,
    q_base_sz, n/2);
  queue.submit([&](cl::sycl::handler &h){
    h.parallel_for({grid_range},
    global_radix_2_ntt_kernel(parameter_list));
  });
}
/* exchanging data via shared local memory */
auto local_grid_range = sycl::nd_range<3>(
  {poly_num, q_base_sz, n/2},\
  {1, 1, WORK_GROUP_SZ});

queue.submit([&](cl::sycl::handler &h){
  h.parallel_for({local_ grid_range},
    slm_radix_2_ntt_kernel(parameter_list));
});
for (; gap > TER_SIMD_GAP_SZ; rounds += m, m *= 2;
    gap /= 2) ;
/* exchanging data via SIMD shuffling */
auto simd_grid_range = sycl::nd_range<3>(
  {poly_num, q_base_sz,\
  n/(2*LOCAL_REG_SLOTS)},\
  {1, 1, WORK_GROUP_SZ});

queue.submit([&](cl::sycl::handler &h){
  h.parallel_for({simd_grid_range},
    simd_radix_2_ntt_kernel(parameter_list));
});
```

Listing 15: SYCL code snippet of the staged NTT

Since the naive radix-2 NTT exchanges data using the global memory, its performance is significantly bounded by the global memory bandwidth. To address this issue, we keep data close to computing units by leveraging shared local memory (SLM) in Intel GPUs, a memory region that is accessible to all the work-items belonging to the same work-group. Here the work-group is analogous to the CUDA thread block. Listing 15 shows the source code to launch SYCL kernels of the staged NTT implementation. Because the data exchanging gap size is halved after each round of NTT, at a certain round, the gap size becomes sufficiently small so that all data to exchange can be held in SLM. We call this threshold gap size `TER_SLM_GAP_SZ`, after which we retain the data in SLM for communication among work-items to avoid the expensive global memory latency. For example, in a 16K-point NTT, we first compute one round of NTT and exchange data using global memory and then the data exchanging gap size has decreased to 4K. We set the `TER_SLM_GAP_SZ` to 4K because the size of the SLM on most Intel GPUs is 64KB, which can hold 8K int64 elements. For the remaining rounds, the data are held in SLM until all computations are completed.

3) SIMD shuffling: In addition to introducing shared local memory, when the exchanging distance becomes sufficiently small that all data to exchange are held by work-items in the same subgroup, we perform SIMD shuffling directly among all the work-items in the same subgroup after NTT butterfly computations. In Fig. 10, we present the rationale of two SIMD shuffling operations among three stages. When the SIMD width equals to 8, there are 8 work-items in a subgroup. For the radix-2 NTT implementation, each work-item holds two elements of the NTT sequence in registers, namely one slot. We denote two local registers of each work-item as Register 0 and Register 1. At the end of Stage 1, where the gap size equals to 8, one needs to exchange data at positions "8, 9, 10, 11" with "4, 5, 6, 7". Such operations can be implemented using `shuffle` of the Intel extension of DPC++ [45]. More specifically, four lanes (lane ID: 0, 1, 2, 3) are exchanging data stored in their Register 1 with Register 0 of lane 4, 5, 6, 7. At the end of Stage 2, where the exchanging gap size is halved from 8 to 4, lanes 0, 1 will exchange data of their Register 1 with Register 0 of lanes 2, 3; similarly, lanes 4, 5 exchange their Register 0 with Register 1 of lanes 6, 7. For the remaining rounds, the data are held in registers and exchanged among work-items in the same subgroup by SIMD shuffling until the gap size becomes equal to 1.

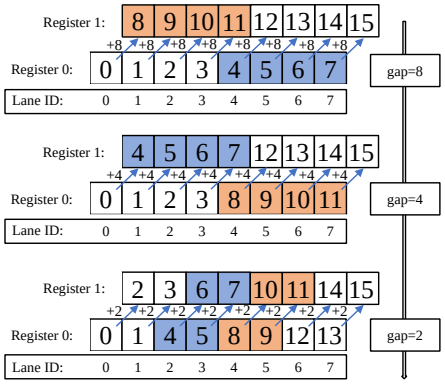


Fig. 10: SIMD shuffling for data exchanging in NTT.

4) More aggressive register blocking: Intel GPUs typically consist of 4KB GRF for each EU thread. When the SIMD width equals 8, that indicates 8 work-items are bounded executing as an EU thread in the SIMD manner. For the radix-2 NTT implementation, each work-item needs four registers, where two of them are used to hold NTT data elements and the other two are for twiddle factors. Therefore, the NTT-related computation consumes $4 \cdot 8 \cdot 8\text{B} = 256\text{B}$ GRF for each EU thread — 6.25% of total GRF, indicating that the hardware is significantly underutilized at the register level.

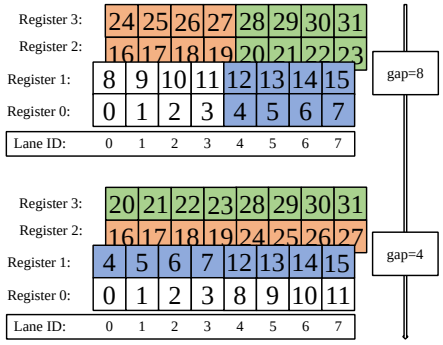


Fig. 11: Multi-slot SIMD shuffling in NTT

Rather than initializing 1 slot of registers, one can assign more workloads (e.g. 2 register slots) to each work-item. For a subgroup of size 8, there are $8 \cdot 2 = 16$ NTT elements being held in registers in the SIMD kernel. We refer it

as SIMD(16,8). Fig. 11 shows a graphical example of the shuffling operation between two stages in SIMD(16,8), while its corresponding source code snippet is presented in Listing 16. In this two-slot SIMD shuffling example, each work-item holds 4 elements in registers, namely 2 slots of registers, for the butterfly computation and data exchanging. Compared with the single-slot implementation, the multi-slot SIMD implementation results in fewer accesses to the shared local memory, but suffers from higher register pressure and the in-register data exchange overhead. In practice, the efficiencies of both 2-slot SIMD(16,8) and 4-slot SIMD(32,8) implementations are worse than the 1-slot SIMD(8,8), suggesting that negative aspects dominate the performance.

```
auto shift_idx = (lane_id >> log_gap);
auto tmp1 = (( shift_idx + 1 ) & 1);
auto tgt_idx = lane_id + (((tmp1<<1)-1) << log_gap);
int slot_idx = 0;
for (; slot_idx < LOCAL_REG_SLOTS; slot_idx++){
  auto reg_idx = tmp1 + (slot_idx << 1);
  data[reg_idx]= sg.shuffle(data[reg_idx], tgt_idx);
}
```

Listing 16: Code snippet of multi-slot NTT SIMD shuffling

5) High-radix NTT: The staged NTT implementation with multi-slot SIMD shuffling increases the register-level data reuse without introducing the register spilling issue. However, this higher hardware utilization on registers comes at a cost of introducing extra efforts to compute target shuffling register indices and lane indices for SIMD shuffling. These integer operations compete for the same ALU port with NTT butterfly computations, adding a non-negligible overhead to the overall performance as shown in Section VI. This performance loss, caused by data exchange overhead, motivates the high-radix NTT implementation, which requires no extra data exchange and communication among work-items compared with the naive radix-2 implementation.

```
// pre-defined macros
// butterfly algorithm to update registers
#define butterfly_ntt_reg2reg(offset_x, offset_y,
    offset_root) \
  u = dwt_guard(dx[offset_x], two_times_modulus);\
  v = dwt_mul_root(dx[offset_y], r_op[offset_root],
    r_quo[offset_root], modulus);\
  dx[offset_x] = dwt_add(u, v);\
  dx[offset_y] = dwt_sub(u, v, two_times_modulus);

// butterfly algorithm to update the global memory
#define butterfly_ntt_reg2gmem(offset_x, offset_y,
    offset_root) \
  u = dwt_guard(dx[offset_x], two_times_modulus);\
  v = dwt_mul_root(dx[offset_y], r_op[offset_root],
    r_quo[offset_root], modulus);\
  *x[offset_x] = dwt_add(u, v);\
  *x[offset_y] = dwt_sub(u, v, two_times_modulus);

// radix-8 NTT internal round 1
butterfly_ntt_reg2reg(0, 4, 0)
butterfly_ntt_reg2reg(1, 5, 1)
butterfly_ntt_reg2reg(2, 6, 2)
butterfly_ntt_reg2reg(3, 7, 3)

// radix-8 NTT internal round 2
butterfly_ntt_reg2reg(0, 2, 0)
butterfly_ntt_reg2reg(1, 3, 1)
butterfly_ntt_reg2reg(4, 6, 2)
butterfly_ntt_reg2reg(5, 7, 3)

// radix-8 NTT internal round 3
// write to the GPU global / shared-local memory
butterfly_ntt_reg2gmem(0, 1, 0)
butterfly_ntt_reg2gmem(2, 3, 1)
butterfly_ntt_reg2gmem(4, 5, 2)
butterfly_ntt_reg2gmem(6, 7, 3)
```

Listing 17: SYCL code snippet of radix-8 NTT

We use radix-8 NTT as an example to demonstrate high-radix NTT algorithms. Each work-item allocates 8 registers to hold NTT elements and 8 more registers to hold root power and root power quotients as for the twiddle factors. For a specific round where the exchanging gap size is gap, one loads eight NTT elements from either global or shared local memory, indexing at $\{k, k+gap, k+2\cdot gap, ..., k+7\cdot gap\}$. There are three internal rounds of butterfly computations before a radix-8 NTT algorithm needs to exchange data among work-items. In the first internal round, four pairs of 2-point butterfly computations are performed among $\{x[k], x[k+4\cdot gap]\}$, $\{x[k+gap], x[k+5\cdot gap]\}$, $\{x[k+2\cdot gap], x[k+6\cdot gap]\}$ and $\{x[k+3\cdot gap], x[k+7\cdot gap]\}$. For the second internal round, these eight elements, still held in registers, are repaired to $\{x[k], x[k+2\cdot gap]\}$, $\{x[k+gap], x[k+3\cdot gap]\}$, $\{x[k+4\cdot gap], x[k+6\cdot gap]\}$ and $\{x[k+5\cdot gap], x[k+7\cdot gap]\}$ so that Algorithm 1 can be leveraged. In the last internal round of the radix-8 kernel, each two consecutive elements are paired and fed into the 2-point butterfly algorithm. After all in-register computations are completed, we store results back to either global memory or shared local memory, depending on whether it is a global memory kernel or a shared local memory kernel. The exchanging gap size *gap* is divided by 8 as a new round of NTT is initiated. Same as the radix-2 NTT, the radix-8 NTT computations are completed when *gap* becomes equal to 1.

Compared with the radix-2 NTT, a radix-R NTT ($R = 4, 8, 16, ...$) decreases the total memory access number from $2N\log_2(N)$ to $2N\log_R(N)$. In coordination with adopting SLM to exchange data, the high-radix NTT maintains the data close to computing units and maximizes the overall efficiency. In Section VI we show that the radix-8 NTT with SLM is up to 4.23X faster than the naive radix-2 NTT on Intel GPUs.

6) The parallelism of staged NTT for HE: Fig. 12 presents the parallelism of NTT in the HE pipeline. Besides mapping one dimensional NTT operations to work-items as elaborated previously, both RNS and the batched number of polynomials can provide the staged NTT implementation with additional parallelisms. To be more specific, the RNS base can be up to several dozens [28] while a batch size can be up to tens of thousands in real-world deep learning tasks [46].

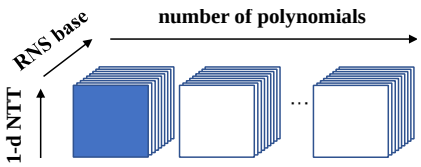


Fig. 12: The parallelism of NTT for HE

C. Application-level Optimizations

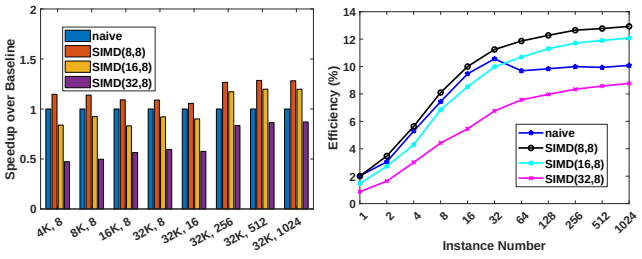
In addition to instruction-level and algorithm-level optimizations, we also optimize our GPU-accelerated HE library from the application level.

VI. Evaluation

We evaluate our optimizations on two Intel GPUs with the latest microarchitecture. Both of them are pre-released models of Intel Xe GPUs. Due to confidentiality requirements, at this time, we do not disclose hardware specifications of these GPUs. For the same purpose, we present performance data by showing normalized execution time rather than the absolute elapsed time or showing performance in the unit of GOPS. The first Intel GPU, denoted as Device1 in following discussions, is a multi-tile GPU while the second Intel GPU, Device2, is a single-tile GPU. We utilize up to 2 tiles in the multi-tile Device1 for performance benchmarking and efficiency estimation. Both GPU devices are connected with 24-core Intel Icelake server CPUs, whose boost frequency is up to 4 GHz. The associated main memory systems are both 128GB at 3200 MHz. We compile programs using Intel Data Parallel C++ (DPC++) Compiler `2021.3.0` with the optimization flag `-O3`.

A. Optimizing NTT on Intel GPUs

Fig. 9 shows that NTT accounts for significant ratios regarding the total execution time of HE evaluation routines. Therefore, we start optimizations from this decisive algorithm.



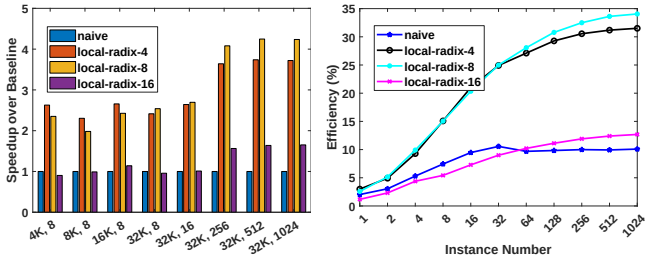
(a) Speedup (b) Efficiency of 32K-point NTT

Fig. 13: Radix-2 NTT with SLM and SIMD on Device1

1) First Trial: optimizing NTT using SLM and SIMD: Fig. 13 (a) compares the speedup of our first batch of NTT trials using the staged NTT implementation over the naive GPU implementation of NTT described in Figure **??**. We use SIMD(`TER_SIMD_GAP_SZ`, `SIMD_WIDTH`) to denote the different implementation variants. Here `TER_SIMD_GAP_SZ` refers to the switching threshold from SLM to SIMD shuffling for data exchanging among work-items. The number of register slots for each work-item can be computed by dividing `TER_SIMD_GAP_SZ` over `SIMD_WIDTH`. For example, each work-item holds a pair of NTT elements in registers for SIMD(8,8), and 4 pairs of NTT elements for SIMD(32,8). With the shared local memory as well as the SIMD shuffling for data exchanging among work-items included, we observe that SIMD(8,8) is faster than the baseline by up to 28%. Meanwhile, SIMD(16,8) is slightly slowed down compared with SIMD(8,8) but remains up to 19% faster than baseline. This indicates that the non-negligible cost of SIMD shuffling leads to the unfavorable performance. Accordingly, SIMD(32,8), which more aggressively performs SIMD shuffling and in-register data exchange than previous two variants, becomes even slower than the baseline.

Figure 13 (a) compares the efficiency of each NTT variant on Device1. The efficiency is computed by dividing the performance of each NTT implementation over the computed int64 peak performance, both in the unit of GFLOPS. The naive NTT reaches only 10.08% of the peak performance for a 32K-point NTT with 1024 instances executed simultaneously. The best one, SIMD(8,8) obtains an efficiency up to 12.93%. Since SIMD shuffling together with the SLM data communication fail to provide us with a high efficiency, we deduce that the cost of data communication is so high that radix-2 NTT cannot fully utilize the device.

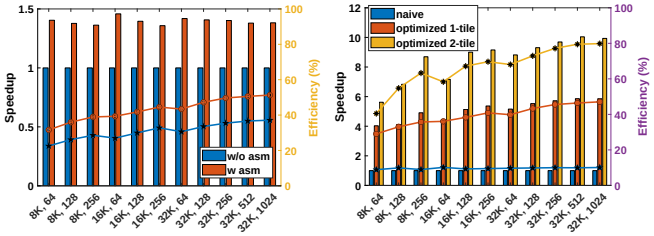
2) Second Trial: optimizing high-radix NTT using SLM: Figure 14 (a) compares the speedup of high-radix NTT implementations with shared local memory against the naive GPU baseline. High-radix NTT implementations re-use more data at the register-level, reducing the communication among work-items through either global memory or SLM. With the shared local memory also included, this time we obtain an up to 4.23X acceleration over the naive baseline. In Figure 14 (b), we see that the efficiency reaches its optimum, 34.1% of the peak performance, at radix-8 NTT with 1024 instances instantiated for 32K-point NTT computations. The radix-16 NTT, though it brings more aggressive register-level data re-use and requires less data exchange among work-items, leads to the register spilling issue so its performance becomes significantly slower than radix-8 NTT.



(a) Speedup (b) Efficiency

Fig. 14: High-radix NTT with SLM on Device1

3) Assembly-level optimizations - `add_mod`/`mul64`*:* We further introduce assembly-level optimizations to improve the speed of the `add_mod` and `mul64` ops. As shown in Figure 15 (a), these low-level optimizations improve the NTT performance by 35.8% - 40.7%, increasing the efficiency of our radix-8 SLM NTT to 47.1%. The inline assembly low-level optimization provides a relatively stable acceleration percentage for different NTT sizes and instance numbers. This is because assembly-level optimization directly improves the clock cycle of the each int64 multiplication and modular addition operation, which is independent of the number of active EUs at runtime.



(a) NTT with inline assembly (b) NTT with explicit dual tile

Fig. 15: NTT with inline-asm and multi-tile on Device1

4) Explicit dual-tile submission through DPC++: With the low-level optimization, we observe that our NTT saturates only up to 47.1%, less than half of the peak performance. We observe this low efficiency because DPC++ runtime does not implicitly support multi-tile execution such that only half of the machine has been utilized. To address this issue, we explicitly submit workloads through multiple queues to enable a full utilization of our multi-tile GPU and manage to reach 79.8% of the peak performance. Meanwhile, our most optimized NTT is 9.93-fold faster than the naive baseline for the 32K-point, 1024-instance batched NTT.

B. Roofline analysis for NTT

The most naive NTT needs to access the global memory for *each* round of the NTT computation. Therefore, its total memory access number can be computed as $2N\log_2(N)$. Here we multiply it by 2 because of both load and store operations at each round of NTT. We do not count the memory access of last round NTT processing to simplify the analysis and because it is negligible.

TABLE I. Number of 64-bit integer ALU operations of each work-item per round for the NTT

	64-bit int ops / round		
	other	butterfly	total
radix-2	20	28	48
radix-4	45	112	157
radix-8	120	336	456
radix-16	260	896	1156

Table I summarizes the number of ALU operations for each NTT variant. *Butterfly* refers to the ALU operations for the NTT butterfly computations while *other* denotes other necessary ALU operations such as index and address pointer computations. The radix-2 NTT performs 48 integer operations for each work-item in a single round of NTT, indicating that the naive NTT consumes $N/2 \cdot 48 \cdot \log_2(N)$ ALU operations throughout the whole computation process. Further dividing the total ALU number over the total memory access number, one can find that the operational density of naive NTT is equal to 1.5 for int64 NTT. This low operational density, as plotted in Figure 16, suggests that the naive NTT implementation is bounded by the global memory bandwidth and can never reach the int64 peak performance.

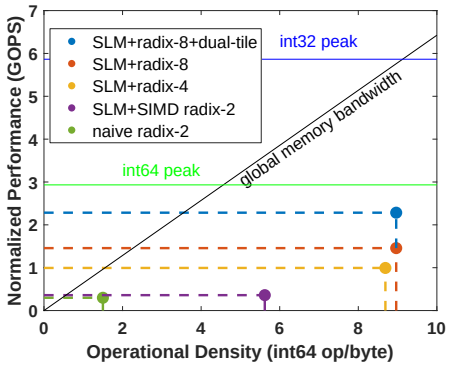


Fig. 16: Roofline Analysis on Device1

For the high-radix NTT, such as the radix-8 implementation for a 32K-point NTT, we first perform one round of radix-8 NTT to reduce the data exchanging gap size from 16K to 2K. After the first kernel stores the data to the global memory, another kernel is launched to compute the remaining rounds of NTT operations, where all the work-groups hold 4K NTT sequence elements in the shared local memory for NTT operations. Therefore, we need only two rounds of global memory access for an instance of 32K-point NTT computation. Considering that its total ALU operation count equals to 456 ALU operations/round $\times \log_8(N)$ rounds — $456 \times \log_8(N)$, one can compute that the operational density of shared local memory radix-8 NTT equals 8.9, pushing the overall performance to the limits of int64 ALU throughput on Device1. The operational density of other NTT variants are computed similarly.

It is worth mentioning that a sound operational density with respect to the global memory access does not guarantee satisfactory overall performance. Although the staged radix-2 NTT with SLM and SIMD shuffling is no longer bounded by the global memory bandwidth, its practical efficiency remains far from the green line — int64 peak performance. According to the Figure 16, we conclude that radix-8 shared local memory NTT with last round kernel fusion enables a sufficient operational density, which allows the performance to be shifted from memory bound to compute bound. Additionally, the shared local memory utilization together with the low-level optimization for int64 multiplication and DPC++ multi-tile submission, pushes the performance of radix-8 NTT to the ceiling of int64 ALU throughput on Device1.

C. Benchmarking for CKKS HE evaluation routines

Figure 17 benchmarks the performance of five basic HE evaluation routines under the CKKS scheme on Device1. Here *MulLin* denotes a multiplication followed by a relinearization; *MulLinRS* denotes a multiplication followed by relinearization and rescaling. Relinearization decreases the length of a ciphertext back to 2 after a multiplication. Rescaling is a necessary step for multiplication operation with the goal to keep the scale constant and reduce the noise present in the ciphertext. In addition, *SqrLinRS* refers to a ciphertext square computation with relinearization and rescaling followed. *MulLinRSModSwAdd*

computes a ciphertext multiplication, then relinearizes and rescales it. After this, we switch the ciphertext modulus from $(q_1, q_2, ..., q_L)$ down to $(q_1, q_2, ..., q_{L-1})$ and scale down the message accordingly. Finally this scaled messaged is added with another ciphertext. The last benchmarked routine, *Rotate*, rotates a plaintext vector cyclically. We count the GPU kernel time exclusively for routine-level benchmarks. All evaluated ciphertexts are represented as tuples of vectors in $\mathbb{Z}_{q_L}^N$ where $N = 32\text{K}$ and the RNS size is $L = 8$.

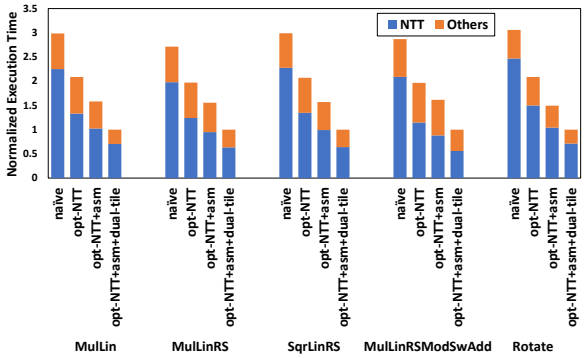


Fig. 17: Benchmarking HE evaluation routines on Device1

We present the impact of NTT optimizations to HE evaluation routines in four steps. We first substitute the naive NTT with our radix-8 NTT with SLM. We then employ assembly-level optimizations to accelerate the clock cycle of int64 *add_mod* and *mul_mod*. Finally we enable implicit dual-tile submission through the OpenCL backend of DPC++ to fully utilize our wide GPU. The baseline is the naive GPU implementation where no presented optimizations are adopted for the comparison purpose.

The radix-8 NTT with data communications through SLM improves the routine performance by 43.5% in average. It is worth mentioning that we do not benchmark batched routines and our wide GPU is not fully utilized such that the NTT acceleration is not as dramatic as the results in previous sections. The inline assembly optimization provides a further average 27.4% improvement in compared with the previous step. Meanwhile, the non-NTT computations show less sensitivity to the inline assembly optimization than the NTT because their computations are typically not as compute-intensive as NTT. We finally submit the kernels through multiple queues to enable the full utilization of our multi-tile GPUs, further improving the performance by 49.5% to 78.2% from the previous step, up to 3.05X faster than the baseline.

D. Benchmarking on Device2

In addition to Device 1, a high-end multi-tile GPU, we benchmark our optimizations on another GPU, Device2, which is a single-tile GPU consisting of fewer EUs than Device1. Similar to the results obtained on Device1, the naive radix-2 NTT starts at a $\sim$15% efficiency of the peak performance, while the shared local memory SIMD implementation either fails to provide significant improvement, but reaches only 20.95%-24.21% efficiencies. After adopting the radix-8 shared local memory implementation, we manage to obtain up to 66.8% of the peak performance, where we are up to 5.47X faster than the baseline at this step. Since Device2 is a single-tile GPU, our final optimization here is to introduce inline assembly to optimize `add_mod` and `mul64`. Further improving the performance by 28.48% in average from the previous step, we reach an up to 85.75% of the peak performance, 7.02X faster than the baseline for 32K-point, 1024-instance NTT.

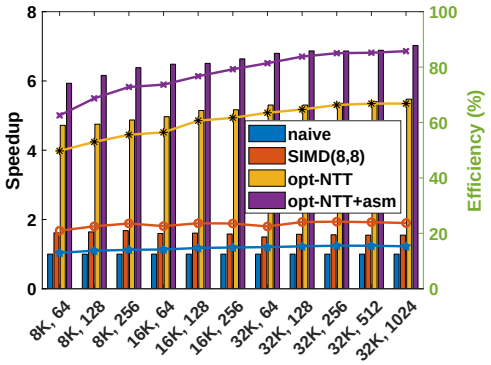


Fig. 18: Benchmark for NTT on Device2

Figure 19 benchmarks the normalized execution time of HE evaluation routines on Device2. SIMD(8,8) denotes the radix-2 NTT with data exchanging through SLM and SIMD shuffling, where each work-item holds one slot of NTT elements in registers. opt-NTT refers to the optimal NTT variant, radix-8 NTT with data exchanging through SLM, shown in Figure 18. The last step is to further employ inline assembly to optimize modular addition and modular multiplication from instruction level. When substituting the naive NTT using SIMD(8,8) NTT, we observe the execution time of the NTT part is improved by 34% in average while the overall routines are accelerated by 29.6%. When switching to our optimal NTT variant, we observe the overall performance becomes faster than the baseline by 1.92X in average. Further enabling assembly-level optimizations, we manage to reach 2.32X - 2.41X acceleration for all five HE evaluation routines on this single-tile Intel GPU.

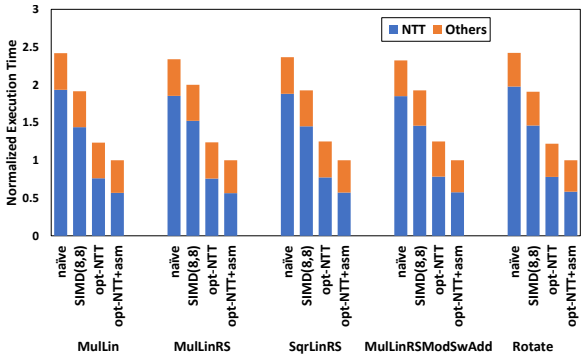
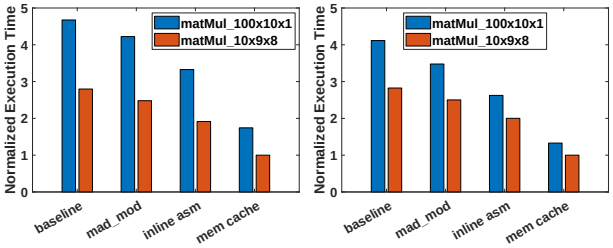


Fig. 19: Benchmarking HE evaluation routines on Device2

E. Benchmarks for polynomial matrix multiplication

Besides the algorithmic level optimizations, we also demonstrate our instruction-level and application-level optimizations,

which are modulus fusion, inline assembly for HE arithmetic operations and memory cache using a representative application of HE, encrypted element-wise polynomial matrix multiplication. In Figure 20, `matMul_mxnxk` denotes a matrix multiplication C+=$A * B$, where C is m-by-n, A is m-by-k and B is k-by-n. Each matrix element is an 8K-element plaintext polynomial so each *element-wise* multiplication of `matMul` is a polynomial multiplication. Modulo operations are always applied at the end of each multiply or addition between polynomial elements. Before starting `matMul`, we need to allocate memory, initialize, encode and encrypt input sequences. Once `matMul` is completed, we decode and decrypt the computing results. We measure the elapsed time for this whole process.



(a) Device 1 (b) Device 2

Fig. 20: Element-wise polynomial multiplication

Figure 20 compares our instruction-level and application-level optimizations for `matMul` on both Device1 and Device2. The fused `mad_mod` and inline assembly accelerate the both 100x100x1 and 10x9x8 polynomial matrix multiplications by 11.8% and 28.2%, respectively, in average on Device1. With the memory cache introduced, both `matMul` applications are further improved by $\sim$90%. On Device1, our all-together systematic optimizations accelerate `matMul_100x10x1` and `matMul_10x9x8` by 2.68X and 2.79X, respectively. In regards to Device2, we observe a similar trend. These three optimizations together provide us with 3.11X and 2.82X acceleration for two `matMul` tests over the baseline on this smaller GPU.

VII. Conclusions

The APIs and algorithms developed in this chapter are sufficient to construct a reasonably complicated computational graph for an application. For example, a logistic regression graph can be constructed using the shown above XeHE methods and computed correctly within the 32-bit floating point precision expectations. As with normal applications of the levelled homomorphic encryption schemes we need to carefully ration the number of multiplies and adjust the budget for levels if necessary. Also there are obvious opportunities in encoding very wide input vectors for computations, since the amount of computations for a single element and 16K elements are same. Encoding thousands of numbers allows to amortize the cost of fully homomorphically encrypted computations.

In this book chapter, we design and develop the first-ever SYCL-based GPU backend for Microsoft SEAL APIs. We accelerate our HE library for Intel GPUs spanning assembly level, algorithmic level and application level optimizations. Our optimized NTT is faster than the naive GPU implementation by 9.93X, reaching up to 85.1% of the peak performance. In addition, we obtain up to 3.11X accelerations for HE evaluation routines and the element-wise polynomial matrix multiplication application. Future work will focus on extending our HE library to multi-GPU and heterogeneous platforms.

We hope a reader may find a few of our ideas helpful in implementing or porting complex DPC++-based applications and libraries.

References

[1] Flexera, https : / / www . flexera . com / blog / cloud / cloud-computing-trends-2021-state-of-the-cloud-report/, Retrieved in 2021, online.

[2] R. L. Rivest, L. Adleman, M. L. Dertouzos et al., "On data banks and privacy homomorphisms," Foundations of secure computation, vol. 4, no. 11, pp. 169–180, 1978.

[3] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in International conference on the theory and applications of cryptographic techniques. Springer, 1999, pp. 223–238.

[4] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." IACR Cryptol. ePrint Arch., vol. 2012, p. 144, 2012.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," ACM Transactions on Computation Theory (TOCT), vol. 6, no. 3, pp. 1–36, 2014.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in International Conference on the Theory and Application of Cryptology and Information Security. Springer, 2017, pp. 409–437.

[7] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2021, pp. 648–677.

[8] C. Gentry, A fully homomorphic encryption scheme. Stanford university, 2009.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption library over the torus," 2016.

[10] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in Proceedings of the 3rd ACM workshop on Cloud computing security workshop, 2011, pp. 113–124.

[11] J. W. Bos, K. Lauter, and M. Naehrig, "Private predictive analysis on encrypted medical data," Journal of biomedical informatics, vol. 50, pp. 234–243, 2014.

[12] J. H. Cheon, M. Kim, and K. Lauter, "Secure dna-sequence analysis on encrypted dna nucleotides," Manuscript at http://media. eurekalert. org/aaasnewsroom/MCM/-FIL, vol. 1439.

[13] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "ngraph-he: a graph compiler for deep learning on homomorphically encrypted data," in Proceedings of the 16th ACM International Conference on Computing Frontiers, 2019, pp. 3–13.

[14] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 325–342.

[15] T. Graepel, K. Lauter, and M. Naehrig, "Ml confidential: Machine learning on encrypted data," in International Conference on Information Security and Cryptology. Springer, 2012, pp. 1–21.

[16] K. Laine, "Simple encrypted arithmetic library 2.3. 1," Microsoft Research https://www. microsoft. com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1. pdf, 2017.

[17] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," IBM Research (Manuscript), vol. 6, no. 12-15, pp. 8–36, 2013.

[18] Y. Polyakov, K. Rohloff, and G. W. Ryan, "Palisade lattice cryptography library user manual," Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep, vol. 15, 2017.

[19] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, V. Gopal et al., "Intel HEXL (release 1.2)," https://github.com/intel/hexl, 2021.

[20] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in International Conference on Cryptography and Information Security in the Balkans. Springer, 2015, pp. 169–186.

[21] nucypher, https://github.com/nucypher/nufhe, Retrieved in 2021, online.

[22] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in International Conference on Selected Areas in Cryptography. Springer, 2018, pp. 347–368.

[23] ——, "Bootstrapping for approximate homomorphic encryption," in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2018, pp. 360–384.

[24] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in International Conference on Cryptology and Network Security. Springer, 2016, pp. 124–139.

[25] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 70–95, 2018.

[26] J.-Z. Goey, W.-K. Lee, B.-M. Goi, and W.-S. Yap, "Accelerating number theoretic transform in gpu platform for fully homomorphic encryption," The Journal of Supercomputing, vol. 77, pp. 1455–1474, 2021.

[27] vernamlab, https://github.com/vernamlab/cuFHE, Retrieved in 2021, online.

[28] S. Kim, W. Jung, J. Park, and J. H. Ahn, "Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus," in 2020 IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2020, pp. 264–275.

[29] S. Kim, K. Lee, W. Cho, Y. Nam, J. H. Cheon, and R. A. Rutenbar, "Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme," in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2020, pp. 56–64.

[30] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 1295–1309.

[31] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: automatically generate high performance dense linear algebra kernels on x86 cpus," in SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2013, pp. 1–12.

[32] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," ACM Transactions on Mathematical Software, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[33] Intel Math Kernel Library. Reference Manual. Santa Clara, USA: Intel Corporation, 2009, iSBN 630813-054US.

[34] Y. Zhai, E. Giem, Q. Fan, K. Zhao, J. Liu, and Z. Chen, "Ft-blas: a high performance blas implementation with online fault tolerance," in Proceedings of the ACM International Conference on Supercomputing, 2021, pp. 127–138.

[35] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Algorithm-based fault tolerance for convolutional neural networks," IEEE Transactions on Parallel and Distributed Systems, 2020.

[36] D. Harvey, "Faster arithmetic for number-theoretic transforms," Journal of Symbolic Computation, vol. 60, pp. 113–119, 2014.

[37] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. Ieee, 2008, pp. 1–12.

[38] Intel Corporation, https : / / software . intel . com / content / dam / develop / external / us / en / documents / the-architecture-of-intel-processor-graphics-gen11-r1new . pdf, Retrieved in 2021, online.

[39] D. Blythe, "The xe gpu architecture," in 2020 IEEE Hot Chips 32 Symposium (HCS). IEEE Computer Society, 2020, pp. 1–27.

[40] Intel, https://github.com/intel/intel-graphics-compiler/tree/master/documentation/visa, Retrieved in 2022, online.

[41] ——, https : / / 01 . org / sites / default / files / documentation / intel-gfx-prm-osrc-icllp-vol02a-commandreference-instructions_2.pdf, Retrieved in 2021, online.

[42] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in 2019 IEEE International symposium on high performance computer architecture (HPCA). IEEE, 2019, pp. 387–398.

[43] W. Jung, E. Lee, S. Kim, K. Lee, N. Kim, C. Min, J. H. Cheon, and J. H. Ahn, "Heaan demystified: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," arXiv preprint arXiv:2003.04510, 2020.

[44] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Springer Nature, 2021.

[45] Intel, https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/SubGroup/SYCL_INTEL_sub_group.asciidoc, Retrieved in 2021, online.

[46] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," arXiv preprint arXiv:1609.04836, 2016.