Accelerating Encrypted Computing on Intel GPUs

Yujia Zhai,[*] Mohannad Ibrahim,[†] Yiqin Qiu,[‡] Fabian Boemer,[‡] Zizhong Chen,[*] Alexey Titov,[‡] Alexander Lyashevsky,[‡]
[*]University of California, Riverside, CA, USA
[†]North Carolina State University, Raleigh, NC, USA
[‡]Intel Corporation, Santa Clara, CA, USA
yzhai015@ucr.edu, mmibrah2@ncsu.edu, yiqin.qiu@intel.com, fabian.boemer@intel.com,
chen@cs.ucr.edu, alexey.titov@intel.com, alexander.lyashevsky@intel.com

***Abstract*—Homomorphic Encryption (HE) is an emerging encryption scheme that allows computations to be performed directly on encrypted messages. This property provides promising applications such as privacy-preserving deep learning and cloud computing. Prior works have been proposed to enable practical privacy-preserving applications with architectural-aware optimizations on CPUs, CUDA-enabled GPUs and FPGAs. However, there is no systematic optimization for the whole HE pipeline on Intel GPUs. In this paper, we present the first-ever SYCL-based GPU backend for Microsoft SEAL APIs. We perform optimizations from instruction level, algorithmic level and application level to accelerate our HE library based on the Cheon, Kim, Kim and Song (CKKS) scheme on Intel GPUs. The performance is validated on two latest Intel GPUs. Experimental results show that our staged optimizations together with optimizations including low-level optimizations and kernel fusion accelerate the Number Theoretic Transform (NTT), a key algorithm for HE, by up to 9.93X compared with the naive GPU baseline. The roofline analysis confirms that our optimized NTT reaches 79.8% and 85.7% of the peak performance on two GPU devices. Through the highly optimized NTT and the assembly-level optimization, we obtain 2.32X - 3.05X acceleration for HE evaluation routines. In addition, our all-together systematic optimizations improve the performance of encrypted element-wise polynomial matrix multiplication application by up to 3.11X.**

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

I. Introduction

The COVID-19 pandemic boosts the rapidly growing demand of enterprises on cloud computing. By 2021, 50% of enterprise workloads are 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 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 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. 5). However, engineering an efficient HE library requires systematical optimizations for the whole HE pipeline beyond computing kernels. In this paper, 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, our contributions include:

- To the best of our knowledge, 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 provide a staged implementation of NTT leveraging shared local memory of Intel GPUs. We also optimize NTT by employing strategies including high-radix algorithm, kernel fusion, and explicit multiple-tile submission.
- From the instruction level, we 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.
- Our 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 paper is organized as follows: we introduce background and related works in Section II, and then detail the asynchronous design and systematic optimization approaches in Section III. Evaluation results are given in Section IV. We conclude our paper and present future work in Section V.

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 paper, 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 $\mathbf{c} = \mathbf{\Psi^{-1}} \odot$ iNTT(NTT($\tilde{\mathbf{a}}$)$\odot$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 paper.

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. Designs and Optimizations

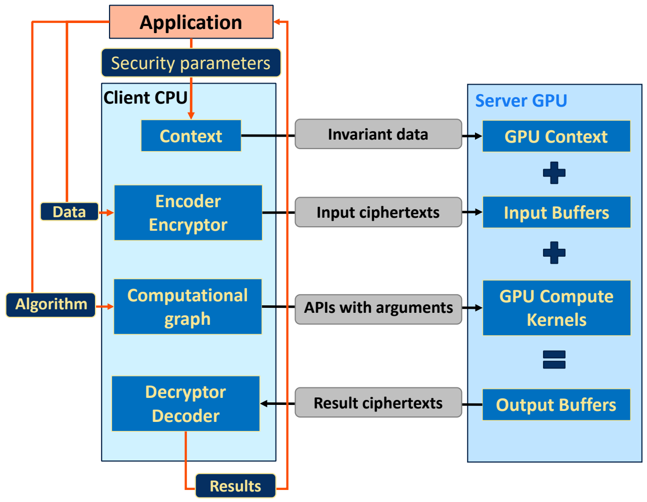


Fig. 1: Client (CPU)/Server (GPU) control/data flow.

In the CKKS scheme, an input message is first encoded and then encrypted to generate ciphertexts using the public key provided by the key generation primitive. We compute directly on the encrypted messages. Once the all computations are completed, the results can be decrypted and decoded by the private key's owner. Figure 1 describes the control flow of our asynchronous HE library. The client host (CPU) generates security parameters, submits GPU compute kernels and sends encrypted data to the GPU upon request. The CPU works asynchronously to the GPU until it receives result ciphertexts from the GPU. Regarding the data flow presented in the same figure, the CPU sends encrypted input data to GPU memory and receives output ciphertexts from the GPU for decryption. Our HE library accelerates the HE evaluation using Intel GPUs while leaving other phases such as key generation, encoding, encryption, decryption and decoding on the client CPU.

Once all the inputs and static data are sent to the GPU, the synchronization with the host becomes unnecessary. Since all host-device synchronizations take additional time, we developed a fully asynchronous execution pipeline to economize on synchronizations. As shown in Figure 2, the computation on

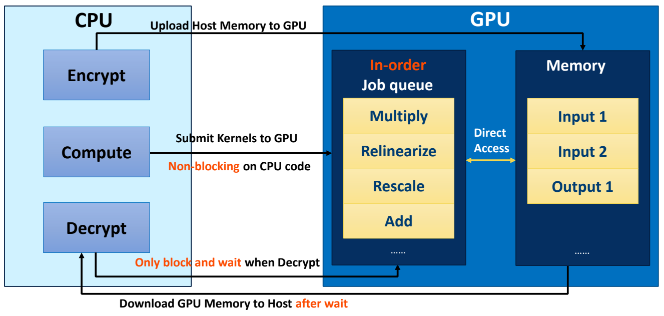


Fig. 2: Asynchronous execution scheme

the GPU starts as soon as the first kernel of the computational graph is submitted. Meanwhile, GPU buffers are allocated and managed at runtime. GPU synchronizes with the host only after the buffers with the results are transferred back to the system memory. In coordination with our memory allocation cache design (Fig. 11), all ciphertext objects are alive until the result ciphertext is sent to the CPU for decryption. At that moment, all differed deletions for memory buffers are served. This design simplifies the lifetime management of ciphertext objects for SEAL API users. In the following contents of this section, we present optimizations of our library from three different angles: instruction, algorithm and application.

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.

2) Optimizing modular addition/multiplication from assembly level: 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, and Unsigned Integer Multiplication.

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. 3: Pseudo int64 addmod assembly

a) Unsigned Modular Addition (`add_mod`*):* Fig. 3(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. 3(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**

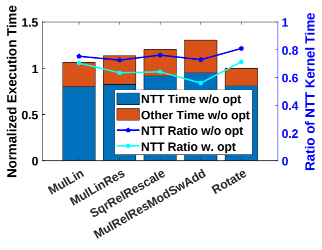
Fig. 4: Pseudo mul64 assembly

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. 4(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.

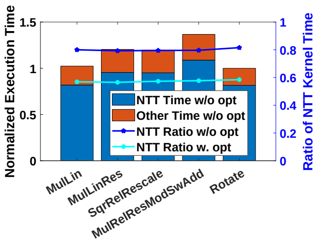
Although the compiler-generated code provides us with a correct result (the lower 64 bits of int64 multiplication), it also computes the higher 64 bits of in64 multiplication, which are redundant in our case. In order to address this issue, we adopt the built-in `mul_low_high` operator to explicitly compute the lower 64-bit multiplication result, as shown in Fig. 4(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 [39].

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 IV, optimizations aimed at our core arithmetic operations greatly impact the performance of HE.

B. Algorithmic level optimizations (NTT)



(a) Profiling on Device1



(b) Profiling on Device2

Fig. 5: Profiling for HE routines

An efficient NTT implementation is crucial for HE computations since it accounts for a substantial percentage of the total HE computation time [28], [40], [41]. Figure 5 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. 16 and 18, 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. This reference implementation of NTT, as shown in Figure 6, 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.

At the lowest level, the NTT butterfly computation 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

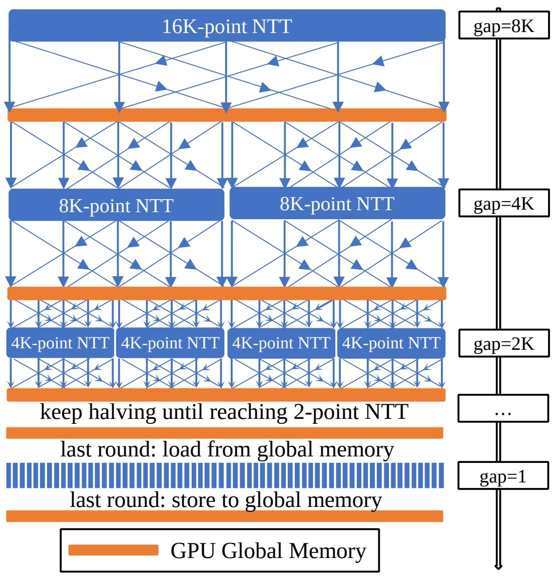


Fig. 6: Naive implementation of 16K-point NTT

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. 12(b). Here the number of instances refers to the number of polynomials in Fig. 10. 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'
```

2) Staged radix-2 NTT with shared local memory: 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. 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 Figure 7, 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++ [42]. 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.

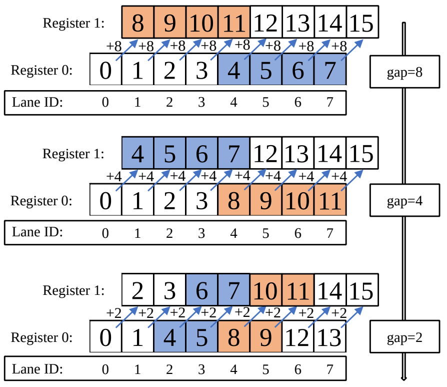


Fig. 7: SIMD shuffling for data exchanging in NTT.

Figure 8 summarizes our staged NTT implementation with both SLM and SIMD shuffling considered using 16K-point NTT as an example. Before the data exchanging gap size reaches the threshold to exchange data in SLM, work-items communicate through the global memory. The gap size is initially equal to 8K and is halved after each round of the NTT butterfly computation. After reaching the SLM threshold (8K-point NTT in this graphical example), one computes NTT butterfly operations and exchanges data through SLM until the gap size equals the threshold to exchange data using SIMD shuffling inside subgroups. It is worth mentioning that the SIMD kernel is fused with the aforementioned last round processing operation to reduce all NTT elements to $[0, p)$.

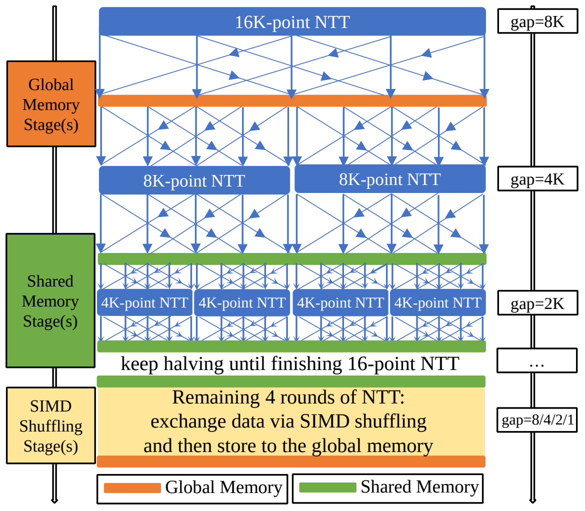


Fig. 8: Staged implementation of 16K-point NTT

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 $4 \cdot 8 \cdot 8$B$=$ 256B GRF for each EU thread — 6.25% of total GRF, indicating that the hardware is significantly underutilized at the register level.

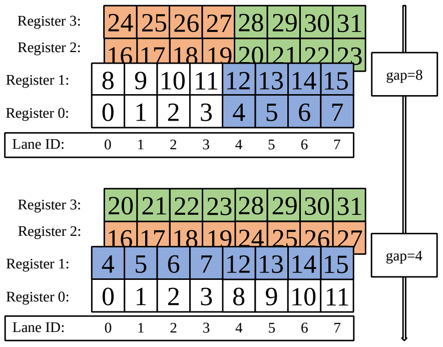


Fig. 9: Multi-slot SIMD shuffling in NTT

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). Figure 9 shows a graphical example of the shuffling operation between two stages in SIMD(16,8). 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.

5) High-radix NTT: The staged NTT implementation with multi-slot SIMD shuffling increases the register-level data re-use 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 IV. 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.

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 re-paired 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 IV 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: Figure 10 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 [43].

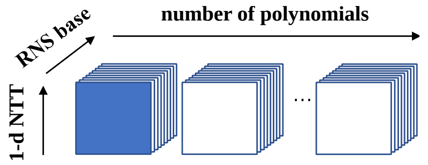


Fig. 10: The parallelism of NTT for HE

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.

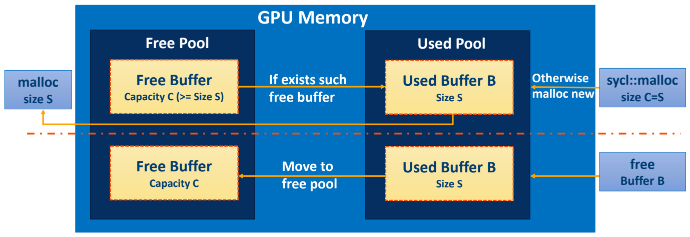


Fig. 11: Memory cache design

1) Memory cache: To reduce the overhead introduced by runtime memory allocation, we design a memory cache mechanism for our HE library, as shown in Figure 11. Similar to Microsoft SEAL, we introduce a memory pool to reuse allocated GPU memory buffers in the HE pipeline. A request for a new GPU memory buffer is routed through the memory cache for any existing free buffer with a capacity larger than the current request. If such buffer is found, this existing buffer is reused instead of allocating a new one. Upon freeing such a buffer, it is moved back to the free pool for potential reuse.

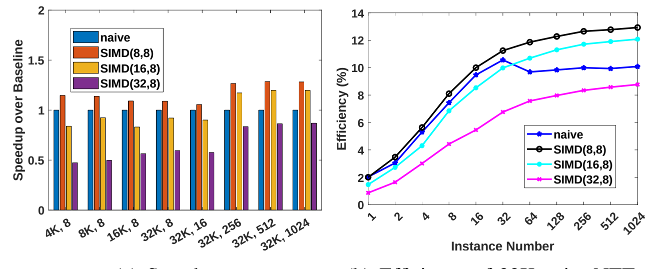
2) Inter-device scaling: Intel packages multiple computing tiles on a single board for scalable performance [44]. At this time, DPC++ does not implicitly support the multi-tile submission. As such, the workloads cannot automatically be distributed over all the computing units of a multi-tile Intel GPU. Therefore, we explicitly submit workloads to the multi-tile device through multiple queues. We refer a reader to [45] for more details of the DPC++ multi-queue submission.

IV. 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

Figure 5 shows that NTT accounts for significant ratios regarding the total execution time of HE evaluation routines. Therefore, we start optimizations from this decisive algorithm.



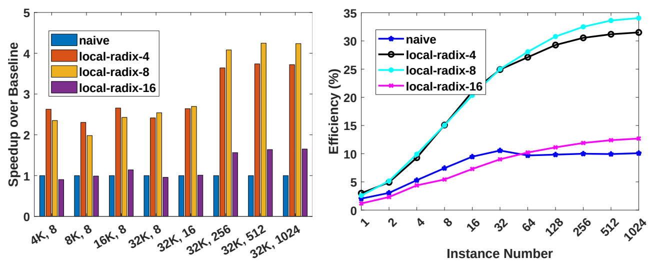
(a) Speedup (b) Efficiency of 32K-point NTT

Fig. 12: Radix-2 NTT with SLM and SIMD on Device1

1) First Trial: optimizing NTT using SLM and SIMD: Figure 12 (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 6. 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 12 (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.

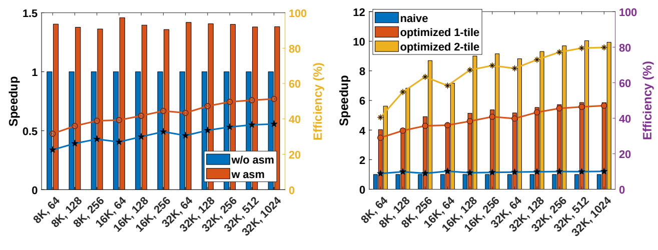
2) Second Trial: optimizing high-radix NTT using SLM: Figure 13 (a) compares the speedup of high-radix NTT implementations with shared local memory against the naive GPU baseline. High-radix NTT implementations re-use more data at the register-level, reducing the communication among work-items through 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 13 (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. 13: 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 14 (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. 14: 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 15, suggests that the naive NTT implementation is bounded by the global memory bandwidth and can never reach the int64 peak performance.

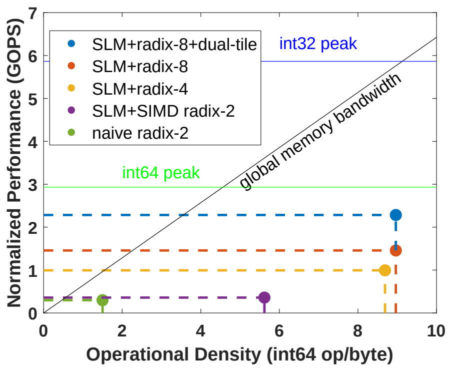


Fig. 15: Roofline Analysis on Device1

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 15, 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 16 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 $N = 32$K and the RNS size is $L = 8$.

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

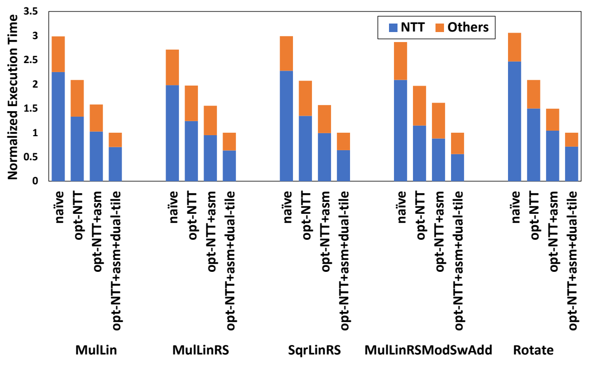


Fig. 16: Benchmarking HE evaluation routines on Device1

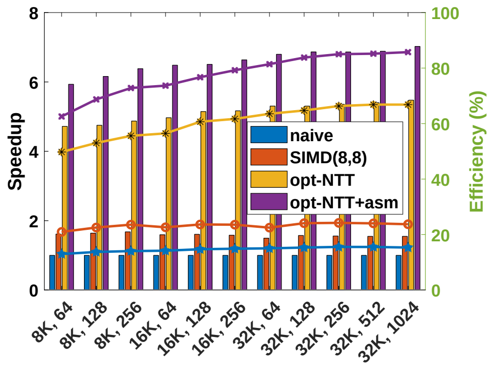


Fig. 17: Benchmark for NTT on Device2

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 server GPU, we benchmark our optimizations on another GPU, Device2, which is a single-tile gaming 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.

Figure 18 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 17. 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.

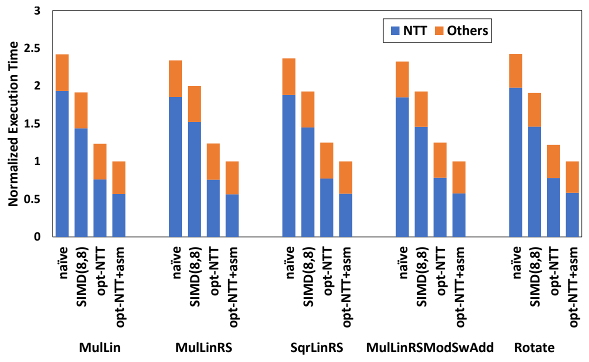
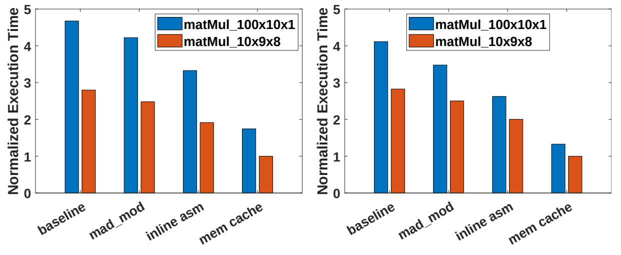


Fig. 18: Benchmarking HE evaluation routines on Device2

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 19, `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. 19: Element-wise polynomial multiplication

Figure 19 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.

V. Conclusions

In this paper, 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.

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

[40] 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.

[41] 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.

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

[43] 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.

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

[45] Intel LLVM, https://intel.github.io/llvm-docs/MultiTileCardWithLevelZero.html, Retrieved in 2021, online.