Intel© Homomorphic Encryption Acceleration Library for FPGAs

Yan Meng,[1*] Fillipe D M de Souza,[1] Hubert de Lassus,[1] Shahzad Butt,[1]
Tomas Gonzales Aragon,[2] Yongfa Zhou,[3] Yong Wang,[3] Flávio Bergamaschi[4*]

Intel Corporation, [1]CA, USA [2]Costa Rica [3]China [4]UK

*To whom correspondence should be addressed; E-mail: he_fpga_support@intel.com.

Abstract

The protection of sensitive data in transit and at rest can be accomplished with traditional data encryption techniques. In this manner, the confidentiality, security and privacy of the data are preserved throughout its life cycle. This strategy requires the data to remain permanently encrypted. With the increasing need to obtain meaningful insights from sensitive data, such as in medical and financial records, so as to generate social and business value, the data must be decrypted for processing. During processing, the data becomes vulnerable to cyberattacks. This violates the protection of the data at all times. In 2009, the seminal work of Gentry gave birth to Fully Homomorphic Encryption (FHE), a technique that allows processing on encrypted data. In particular, FHE enables computation of arbitrary functions on encrypted data without ever decrypting it. The benefits of FHE come at the cost of high computational demand. The performance of algorithms evaluating functions on encrypted data is many orders of magnitude lower than their counterparts computing on unencrypted data. Narrowing this performance gap is key to unlock the capability of data privacy protection during computation, pervasively. We propose FPGAs compute architectures that can be used to obtain throughput performance gains in critical Homomorphic Encryption (HE) operations.

Privacy-Preserving Computing

The concept of privacy-preserving computing arises when two or more organization wish to benefit from a collaboration that involves sharing and processing of private data whose ownership do not intersect. Fraud detection and more accurate medical diagnosis are common applications that could benefit from collaborative work involving private data. In practice, this is often impeded for competitive reasons and legal regulation requirements. Homomorphic Encryption is an encryption technique that can address this problem.

Homomorphic Encryption

Homomorphic Encryption (HE) is a type of encryption technique that enables computation whilst data remains encrypted. This type of encryption scheme is built on the hardness of the mathematics of lattice problems (*4*) (*6*). Lattice-based cryptographic constructions that rely on the worst-case hardness of lattice problems offer strong provable post-quantum computing security guarantees (*11*) (*10*); thus, useful to create secure cryptosystems.

In 2009, the seminal work of Gentry (*5*) put forward the concept of bootstrapping, giving birth to practical Fully Homomorphic Encryption (FHE) implementations that allow processing of arbitrary functions on encrypted data. This capability has the potential to unlock a plethora of use cases to run on untrusted cloud computing infrastructures without violating data privacy and confidentiality requirements. Over the years, many types of HE schemes and HE software libraries implementing these schemes have been proposed. The HE software libraries provide APIs for application developers building privacy-preserving workloads. Microsoft SEAL (*13*), HELib (*7*) and PALISADE (*1*) are among the most popular open-source HE software libraries. They differ in implementation, performance and support of the types of HE schemes. However, they all carry something in common. Under the hood, their code implements algorithms that perform heavy modular polynomial arithmetic operations on large amounts of data. These operations and the size of the encrypted data (ciphertexts) form the basis of the compute bottlenecks for HE-based workloads.

Vision Statement

HE technology has the potential to transform the way protection of data privacy and confidentiality is handled in the digital data domain. With Intel HEXL for FPGA, Intel wishes to contribute to the development of a hardware and software ecosystem that supports deployment of HE-driven applications at large scale. We welcome collaborative efforts with both the academic and industry communities. To this end, we publish a preliminary work, under an open-source license, that attempts to bring up FPGAs as co-processors to improve throughput of HE operations. With this initiative, we invite both communities to design and co-engineer the future of privacy-preserving technology using FPGAs with us.

Intel HEXL for FPGA

We introduce Intel© Homomorphic Encryption Acceleration Library for FPGAs (Intel HEXL for FPGA), an open-source collection of FPGA kernels for common computing building blocks of Homomorphic Encryption (HE). More especifically, the purpose is to provide the community of HE developers with a baseline implementation of FPGA processing elements for homomorphic encryption arithmetic built on lattice-based cryptography constructions (*4, 10, 11*). In general, operations of lattice-based cryptosystems involve heavy modular polynomial arithmetic, including *element-wise polynomial additions*, *element-wise polynomial subtractions*, *element-wise polynomial multiplications* and *polynomial-polynomial multiplications*. HEXL-FPGA provides FPGA kernel implementations to handle these types number-theoretic polynomial operations. These kernels were designed to improve system-level throughput performance of privacy-preserving applications based on HE. As follows, we provide a brief overview of the basic mathematics of modular polynomial arithmetic used in lattice cryptography and an explanation of their implementations as FPGA kernels.

Modular Polynomial Arithmetic

In lattice cryptography, the common building blocks typically operate on the domain of polynomial quotient rings $R = Z/qZ[X]/(X^N + 1)$. Those polynomials of at most N-1 degree have their coefficients as integers in the finite field $Z/qZ = \{0, 1, ..., q - 1\}$, where q denotes a multi-precision prime number, N is conventionally a power of 2 and the $q \equiv 1 \mod 2N$ equation is satisfied. The parameters N and q form the basis of the security level strength. Finally, the encrypted data consists of a pair of polynomials $c = (a, b) \in R_q^2$, where c is refered to as a ciphertext.

Therefore, the common algorithmic building blocks of lattice cryptography involve polynomial ring arithmetic, including modular polynomial additions and multiplications. Polynomials are typically represented as a vector of coefficients $a = \{a_0, a_1, ..., a_{N-1}\}$. A ciphertext-ciphertext addition consists of element-wise vector additions and a ciphertext-ciphertext multiplication turns into a tensor product, *i.e.* $c \otimes c' = (a, b) \otimes (a', b') \rightarrow (a \odot a', a \odot b' + b \odot a', b \odot b') \in R_q^3$, where $\otimes$ and $\odot$ correspond to tensor product and element-wise product, respectively. The naïve implementation of this operation has $O(N^2)$ computational complexity.

In order to perform ciphertext-ciphertext multiplication more efficiently, *i.e.* in $O(N \log N)$ asymptotic complexity, the number-theoretic transform (NTT) algorithm is often employed. NTT is fast Fourier Transform (FFT) over a finite field of integers, such that additions and multiplications are performed with respect to the modulus q. It is one of the most critical performance bottlenecks in lattice-based encryption schemes. Much work has been proposed to speed up its performance (*2,8*), including with accelerators such as GPUs (*9*) and FPGAs (*3,12*).

Note that the resulting ciphertext is in R_q^3, which means that an additional operation, named *relinearization*, migth still be needed in order to continue the sequence of ciphertext operations. *Relinearization* moves the resulting ciphertext of a ciphertext-ciphertext multiplication back to R_q^2. The support for the *relinearization* operation will be published in a later release when the *KeySwitch* operation, on which it depends, will also be available – a more detailed explanation of these operations is out of the scope of this document.

FPGA Kernels

Below we list the operations used in homomorphic encryption that are provided in the Intel HEXL for FPGA open-source release.

- *Forward and Inverse NTT* Both the forward and inverse implementations of the number-theoretic transform algorithm, NTT and INTT, respectively. NTT is equivant to a Discrete

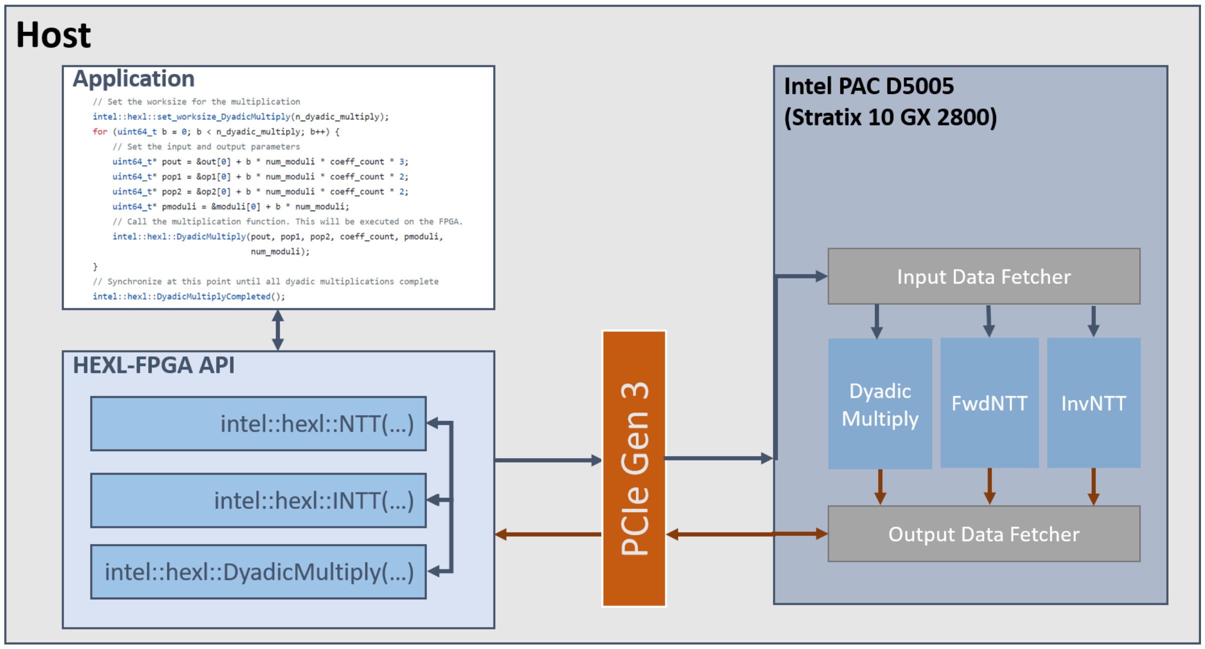
Fourier Transform (DFT) over a finite field of integers (*3, 9*). In our implementation, twiddle factors are kept constant and loaded just once. NTT API calls can be synchronous (blocking and slow), as well as, asynchronous for a specific code block where calls are made to offload NTT (or INTT) work only. Ansyncronous calls are possible provided that the same set of twiddle factors are reutilized in all the calls. The ciphertexts subject to NTT processing are internally buffered and batched for processing in the FPGA.

- *Dyadic Multiplication* To multiply two ciphertexts, we assume they have been transformed into their NTT form so as to perform an NTT-based polynomial multiplication, *i.e.* the tensor product $\hat{a} \otimes \hat{b}$, where $\hat{a} = NTT(ct_a) = (NTT(a_0), NTT(a_1))$ and $\hat{b} = NTT(ct_b) = (NTT(b_0), NTT(b_1))$. The dyadic multiplication API calls can also be synchronous or asynchronous. In the asynchronous case, the same rules as for NTT's apply. The resulting ciphertext has three components and needs to be relinearized on the host side by the programmer.

Fig. 1 gives an overview of the contents of this release, the software stack and briefly how it is used. A host application makes asynchronous Intel HEXL for FPGA API calls inside a specific code block defined by a *for* loop. Multiple asynchronous calls to the dyadic multiplication operation are issued. Input data for each dyadic multiplication resides in a contiguous memory space. In the inner body of the *for* loop, `intel::hexl::DyadicMultiply(...)` work requests are enqueued in a buffer and sent to the FPGA in smaller chunks of data that called batch. At the end of the *for* loop code block, the program waits until all the work for the set of calls has been completed, where `intel::hexl::DyadicMultiplyCompleted(...)` works as an explicit barrier. On the device side, a batch of data is pulled and delivered for processing to the target kernel. The output data fetcher waits for the batch processing completion before sending the results back to host the via PCIe. Since this mechanism is asynchronous, data movement overlaps as much as possible with FPGA compute work.

Optimizations

We briefly describe the optimizations employed in order to get the most of the kernels performance on batched lazy execution.



```
// Set the worksize for the multiplication
intel::hexl::set_worksize_DyadicMultiply(n_dyadic_multiply);
for (uint64_t b = 0; b < n_dyadic_multiply; b++) {
    // Set the input and output parameters
    uint64_t* pout = &out[0] + b * num_moduli * coeff_count * 3;
    uint64_t* pop1 = &op1[0] + b * num_moduli * coeff_count * 2;
    uint64_t* pop2 = &op2[0] + b * num_moduli * coeff_count * 2;
    uint64_t* pmoduli = &moduli[0] + b * num_moduli;
    // Call the multiplication function. This will be executed on the FPGA.
    intel::hexl::DyadicMultiply(pout, pop1, pop2, coeff_count, pmoduli,
                                num_moduli);
}
// Synchronize at this point until all dyadic multiplications complete
intel::hexl::DyadicMultiplyCompleted();
```

Figure 1: Intel HEXL for FPGA System View

System Level

Intel HEXL for FPGA API was built based on a pull model to provide best throughput and scalability. High system level throughput is achieved through non-blocking single-type API calls for a dedicated section of code that asynchronously overlap pre-/post-computation, I/O data transfer and kernel computation. The design consisted of profiling-guided optimizations, host double buffering and bursting memory accesses.

Kernel Level

On the device side, parallel execution is achieved with loop-unrolling (data parallelization) and functional concurrency with deep pipelining (loop pipelining). Better latency and data locality were achieved with data prefetching, memory coalescing, BRAM banking, bank conflicts resolution through data reordering and replication, data packing and data reuse.

Learnings

- System-level optimization is critical to improve performance throughput, especially for I/O bound compute building blocks;
- There is a per-case performance trade-off between leaving some input data as constant versus computing them on-the-fly;
- Workloads pipeline profiling is required in order to get the most of FPGA computational performance and resources.

Advantages of FPGAs

FPGAs offer a flexible and programmable platform to design and implement power efficient and high-throughput compute architectures. It also provides opportunity to leverage as much functional and data parallelism as the structure of the target algorithm presents. Additionally, it can execute on streaming mode such that data movement perfectly overlaps with compute. All of those characteristics combined make FPGAs suitable co-processors to accomplish more work per unit of time. As a result, it has the potential to improve overall throughput performance of HE workloads at the system level. We summarize its advantages below.

- Architecture design flexibility;
- Enables streaming-mode and pipelined execution;
- Low power consumption per performance gain;
- Friendly to exploit data and functional parallelism to increase throughput;
- Easily adaptable and programmable to ever-changing algorithm implementations.

Getting Started

The Intel HEXL for FPGA github repository is located at `https://github.org/intel/hexl-fpga`. It hosts the source codes of FPGA kernels written and designed with the Intel FPGA SDK for OpenCL 2.0, namely, the *dyadic multiplication* operation for ciphertext-ciphertext multiplication, the forward and inverse *number-theoretic transforms* (NTT). It also provides modular arithmetic operations as reusable Intellectual Properties (IP) modules created with the Intel High-Level Synthesis (HLS) tool. Their high-level designs are then synthesized and their FPGA bitstreams generated using the Intel Quartus Prime tool. Those IP modules are utilized in the FPGA kernel implementations to compute elementwise vector arithmetic. These kernels were validated on a system hosting the Intel PAC D5005 (Stratix 10 GX 2800, such as in (*12*)) PCIe card. Additionally, host side support is provided through a C/C++ FPGA Runtime API that can be used to actually launch those kernels and execute them in the FPGA chip. The source codes are published under the Apache License 2.0. In Fig. 2, we give a pictorial overview of the components that make up the Intel HEXL for FPGA package. We further describe those components in Table 1. Download it now and check out the README.md for a quick start.

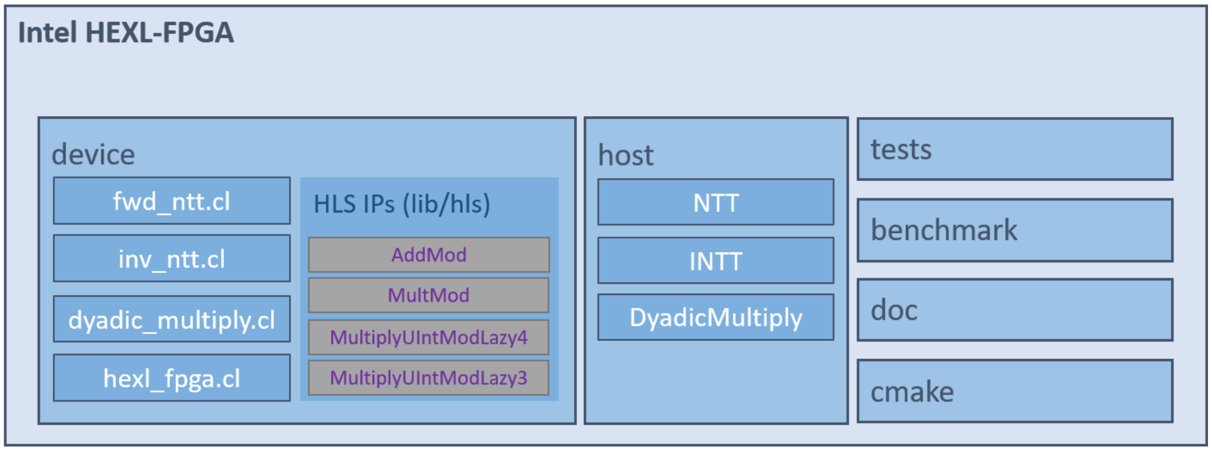


Figure 2: Intel HEXL for FPGA Package Overview

License

The Intel HEXL for FPGA repository release is copyrighted by Intel and distributed under Apache 2.0 license. Any reuse and redistribution of the repo contents should abide by the Apache 2.0 license agreement. See the LICENSE file for details.

Table 1: Description of Intel HEXL for FPGA components.

`cmake`	Support files to the CMake build system to include `gtest` and `gbenchmark` frameworks as dependencies for tests and benchmarks of Intel HEXL for FPGA.
`doc`	Source files to build the Intel HEXL for FPGA Runtime API documentation.
`benchmark`	A set of benchmarks for each specific FPGA kernel using the host FPGA runtime API.
`tests`	A set of examples that test the usage and correctness of the FPGA kernels using the host FPGA runtime API.
`host`	Source code of the host FPGA runtime API.
`device`	Source code of the OpenCL-based FPGA kernels and reusable HLS-based FPGA IP modules.

Contributions and Future Work

Future directions include the distribution of all the currently provided kernels ported to Intel OneAPI, a unified programming paradigm for heterogeneous computing, and additional functionalities to support more homomorphic encryption operations, such as relinearization and rotation. The community of HE practinioners, from academia to industry, is invited and welcomed to contribute to this effort with new optimizations, additional FPGA kernels, as well as, by extending the C/C++ HEXL for FPGA Runtime API. Learn more on how to contribute to this project by reading CONTRIBUTING.md.

References

1. PALISADE Lattice Cryptography Library (release 1.11.2). `https://palisade-crypto.org/`, May 2021.

2. Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). `https://github.com/intel/hexl`, 2021.

3. Donald Donglong Chen, Nele Mentens, Frederik Vercauteren, Sujoy Sinha Roy, Ray C. C. Cheung, Derek Pao, and Ingrid Verbauwhede. High-speed polynomial multiplication ar-

chitecture for ring-lwe and she cryptosystems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 62(1):157–166, 2015.

4. Dong Pyo Chi, Jeong Woon Choi, Jeong San Kim, and Taewan Kim. Lattice based cryptography for beginners. *IACR Cryptol. ePrint Arch.*, 2015:938, 2015.

5. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

6. Craig Gentry. Toward basing fully homomorphic encryption on worst-case hardness. In *Annual Cryptology Conference*, pages 116–137. Springer, 2010.

7. Shai Halevi and Victor Shoup. Design and implementation of helib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. `https://ia.cr/2020/1481`.

8. David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.

9. Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 264–275. IEEE, 2020.

10. Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

11. Hamid Nejatollahi, Nikil Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. Post-quantum lattice-based cryptography implementations: A survey. *ACM Computing Surveys (CSUR)*, 51(6):1–41, 2019.

12. M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei 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*, pages 1295–1309, 2020.

13. Microsoft SEAL (release 3.7). `https://github.com/Microsoft/SEAL`, September 2021. Microsoft Research, Redmond, WA.