Towards Full-Stack Acceleration for Fully Homomorphic Encryption

Naifeng Zhang* Homer Gamil† Patrick Brinich‡ Benedict Reynwar§ Ahmad Al Badawi¶ Negar Neda† Deepraj Soni† Kellie Canida§ Yuriy Polyakov¶ Patrick Broderick‖ Michail Maniatakos† Andrew G. Schmidt§ Mike Franusich‖ Jeremy Johnson‡ Brandon Reagen† David Bruce Cousins¶ and Franz Franchetti*

*Carnegie Mellon University {naifengz, franzf}@cmu.edu, ‡Drexel University {pjb338, johnsojr}@drexel.edu

†New York University {homer.g, nn2231, dss545, mm6446, bjr5}@nyu.edu

§USC Information Sciences Institute {breynwar, kcanida, aschmidt}@isi.edu

¶Duality Technologies {aalbadawi, ypolyakov, dcousins}@dualitytech.com

‖SpiralGen, Inc. {patrick.broderick, mike.franusich}@spiralgen.com

I. Introduction

Fully Homomorphic Encryption (FHE) [1] allows direct computation on sensitive information through mathematical transforms upon encrypted data. FHE uses lattice-based cryptography to encode vectors of numerical data onto mathematical structures (lattices and rings), thereby enabling a set of basic arithmetic operations to be performed while encrypted. These operations can be combined for applications like pattern matching, linear algebra, basic statistics and machine learning.

To encode and operate on different types of data, several schemes have been proposed, including BGV [2], BFV [3], and CKKS [4]. All these schemes are based on lattice cryptography, requiring a core set of mathematical operations in the form of integer modulo vector arithmetic.

Adoption of FHE requires high-performance implementation of these schemes including special-purpose hardware. A major optimization for FHE schemes is using Number Theoretic Transform (NTT) to accelerate the computation of vector convolutions, similar to the fast Fourier Transform (FFT) in the signal processing domain. High-performance NTTs require automatic code generation and autotuning similar to the FFT. However, NTT performance on standard hardware is relatively poor, and automating NTTs is not as well-studied. Thus, we approach the problem by building an end-to-end accelerator for FHE, with a focus on automatic NTT generation for specialized hardware.

Contributions. Our key contributions are:

- *Top layer.* Generating PALISADE homomorphic encryption library.
- *Code generator.* Introducing NTTX: SPIRAL's NTT library in the vein of FFTW/FFTX.
- *Bottom layer.* Introducing TILE: a special FHE hardware.
- Demonstrating that the 64K-point NTT generated by the PALISADE-NTTX-TILE system results in 9.161us on Palladium emulator.

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-20-S0032. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

II. Approach

A. PALISADE

Several popular, high quality, open source libraries exist for implementing systems based on FHE. SEAL from Microsoft Research [5], HELib from IBM research [6] and PALISADE [7] by multiple authors on this paper from Duality Technologies. We chose PALISADE to implement our work, as it supports all the mentioned schemes (along with multiple variants to allow multiple parties to share and compute on data without revealing it), is implemented in C++ and is heavily optimized for vector operations, using residue arithmetic to reduce large bitsize arithmetic into smaller conventional machine words.

The PALISADE library follows the Homomorphic Encryption Standard by meeting proper bit-security thresholds, given via (semi-)automated parameters that can be set by the user to control what security and performance they prefer to target.

B. TILE

Our accelerator implements a custom vector Instruction Set Architecture (ISA) tailored to the needs of FHE. This was done to meet the objective of a flexible design (i.e., not fixed hardware) while delivering orders of magnitude of performance better than GP-CPUs. The instruction set uses 512 length vectors and 128-bit native words. Long vectors are used to amortize the overhead of programmable computing (e.g., instruction fetch and decode) and to decouple the architecture from the microarchitecture, as vectors allow us to scale up or down parallel resources to tradeoff area and performance. The ISA has notable stateful structures for vector data, scalar data, the vector Register File (RF), a modulus RF, and the address RF. Instruction support is kept intentionally minimal, each is added only if it is vital for functionality or performance. The current ISA includes only 17 unique instructions, which range in complexity from *LOAD* and *ADD* to *BFLY* and *SHUFFLE*.

The TILE microarchitecture is designed for high performance and efficiency by decoupling all pipelines and keeping non-compute logic simple, maximizing chip resources devoted to the actual computation. The frontend of the machine constitutes simple in-order logic and tracks dependent instructions via dynamic register dependence detection. Instructions are not issued to the backend until they are determined to be independent. The backend consists of three unique pipelines: compute, memory, and (register-register) shuffle. Since all issued instructions are independent, the pipelines are free to execute in parallel.

C. NTTX

We extended SPIRAL [8] to support NTT and batch NTTs. Mirroring the structure of FFTW and FFTX, the NTTX package offers FFTW-style C/C++ API in line with FFTX-style code generation, powered by SPIRAL in the backend. Illustrated by Fig. 1, NTTX API leverages SPIRAL's capability of delayed execution and just-in-time code generation to implement an inspector/executor paradigm for PALISADE.

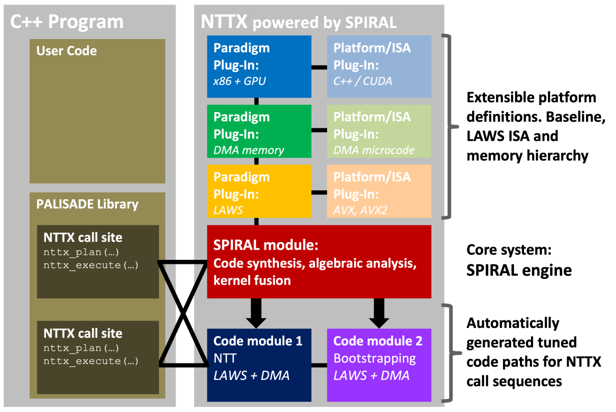
C++ Program
User Code
PALISADE Library
NTTX call site
nttx_plan(…)
nttx_execute(…)
NTTX call site
nttx_plan(…)
nttx_execute(…)
NTTX powered by SPIRAL
Paradigm Plug-In: x86 + GPU
Platform/ISA Plug-In: C++ / CUDA
Paradigm Plug-In: DMA memory
Platform/ISA Plug-In: DMA microcode
Paradigm Plug-In: LAWS
Platform/ISA Plug-In: AVX, AVX2
SPIRAL module: Code synthesis, algebraic analysis, kernel fusion
Code module 1 NTT LAWS + DMA
Code module 2 Bootstrapping LAWS + DMA
Extensible platform definitions. Baseline, LAWS ISA and memory hierarchy
Core system: SPIRAL engine
Automatically generated tuned code paths for NTTX call sequences

Fig. 1. Overview of the interaction between NTTX and PALISADE.

As shown in Listing 1, the NTTX package can be exported as a C/C++ library with NTTX API for general usage.

```
// C/C++ NTTX API example: compute a singel NTT
#include "nttx.h"
nttx_plan *p;
p = nttx_plan_ntt(in, out, n, modulus, NTTX_FORWARD);
nttx_execute(p);
nttx_free(p);
```

Listing 1: NTTX C/C++ API.

To support general radix NTTs, large vector instructions and simple parallelism in SPIRAL, we added both the Korn-Lambiotte FFT algorithm [9] and the Pease FFT algorithm [10] as breakdown rules to SPIRAL. Using SPIRAL's Operator Language (OL), NTTs of size r^k are represented as

$$\mathrm{NTT}_{r^k} = \mathrm{R}_r^{r^k} \left(\prod_{i=0}^{k-1} \mathrm{L}_{r^{k-1}}^{r^k} \, \mathrm{D}_i^{r^k} \, (\mathrm{NTT}_r \otimes \mathrm{I}_{r^{k-1}}) \right)$$

To execute the generated NTT code on TILE, NTTX allows various data types for long vectors, provides different schemes of register allocation (e.g., greedy, naive round robin), and has the infrastructure for verification (e.g., functional simulator) and low-level optimizations (e.g., instruction scheduler).

We generated forward and inverse vectorized radix-2 NTTs with sizes from 1,024 to 65,536, and verified their correctness with PALISADE data. Listing 2 shows the radix-2 1,024-point NTT code generated by SPIRAL.

```
// SPIRAL generated NTT Code for TILE vector architecture
#include <tile.h>
void _ntt1024x512_b1() {
    enter(OP_DEFAULT);
    _vload_512x128i(REG_V60, REG_A1, 0);
    _vload_512x128i(REG_V20, REG_A1, 8192);
    _vbroadcast_512x128i(REG_V19, REG_A3, 1, 1);
    _vimulmod_512x128i(REG_V59, REG_V20, REG_V19, REG_M1);
    _vaddmod_512x128i(REG_V58, REG_V60, REG_V59, REG_M1);
    _vsubmod_512x128i(REG_V57, REG_V60, REG_V59, REG_M1);
    _vunpacklo_512x128i(REG_V56, REG_V58, REG_V57);
    ...
    _vstores_512x128i(REG_A2, 16, REG_V21, 2);
    leave(OP_DEFAULT);
}
```

Listing 2: SPIRAL-generated radix-2 1,024-point NTT code using shuffle instructions.

III. Results

We utilized Air Force Research Laboratory (AFRL)'s Palladium emulation system to confirm performance of kernels and operations running on the Ring Processing Unit (RPU). The Palladium platform allowed the team to verify the functionality and confirm our cycle accurate counts for each kernel across the simulation and testing platforms. Emulated on Palladium using 512-way vectors at 2 GHz, 128-bit scalar and vector add/sub/mul takes 8 clock cyles (4ns) and the 64K-point radix-2 128-bit NTT takes 18,322 clock cycles (9.161us).

IV. Conclusion

This paper provides a first look at the end-to-end FHE accelerator, which is optimized by PALISADE on the algorithmic level, by NTTX from SPIRAL on the code generation level, by TILE on the microarchitecture level. Our work exhibits the necessary structure and components for an integrated end-to-end system for FHE acceleration.

References

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

[2] 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.

[3] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[4] 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*, pp. 409–437, Springer, 2017.

[5] "Microsoft SEAL (release 4.0)." https://github.com/Microsoft/SEAL, Mar. 2022. Microsoft Research, Redmond, WA.

[6] S. Halevi, "Helib (version 2.1.0)," 2021.

[7] P. team, "PALISADE Lattice Cryptography Library (release 1.11.6)," 2022.

[8] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[9] D. G. Korn and J. J. Lambiotte, "Computing the fast fourier transform on a vector computer," *Mathematics of computation*, vol. 33, no. 147, pp. 977–992, 1979.

[10] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.