Generating Number Theoretic Transforms for Multi-Word Integer Data Types

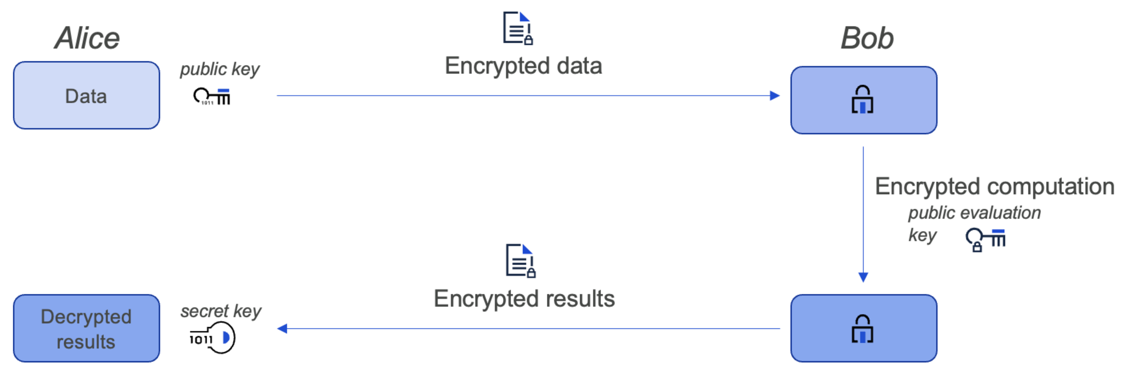
Naifeng Zhang, Franz Franchetti; *Carnegie Mellon University*

IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2023

Carnegie Mellon University

Fully Homomorphic Encryption

- Fully Homomorphic Encryption (FHE) serves as a cryptographic approach that allows cloud platforms to manipulate encrypted data.



FHE Scheme. Credit: Duality Technologies.

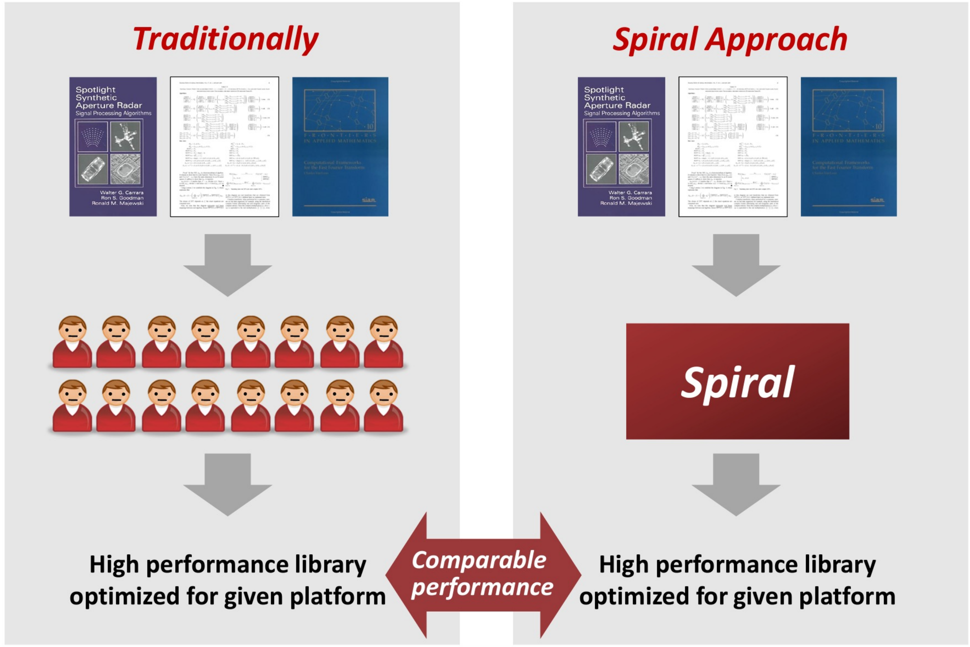
- Yet, a significant amount of computing power and time is required by FHE, where the bottleneck resides in polynomial multiplication.

Number Theoretic Transform

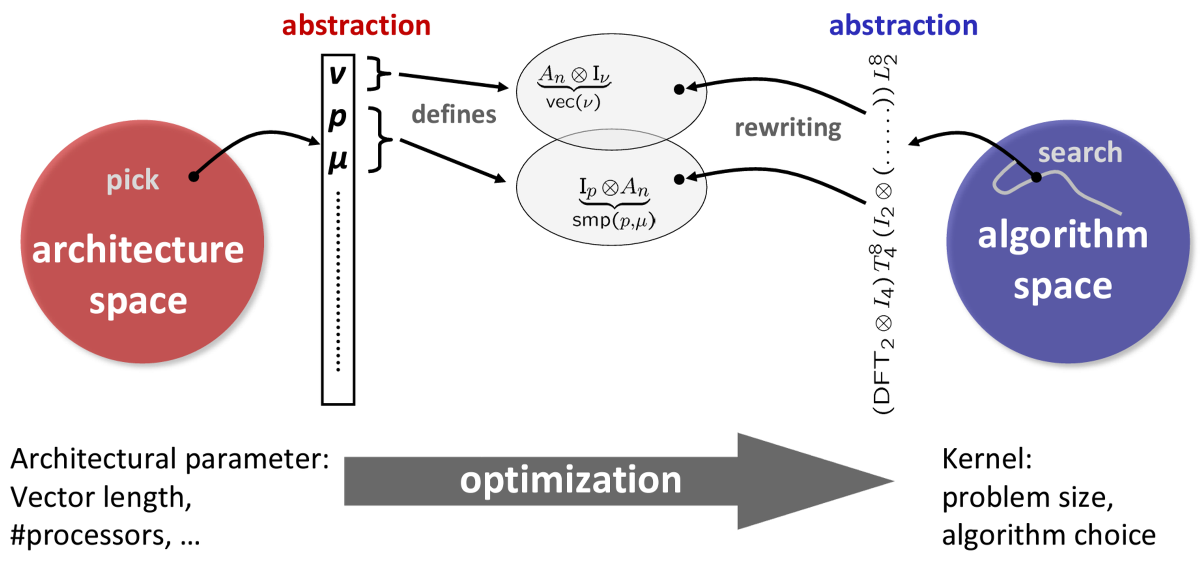
- Number Theoretic Transform (NTT) is a popular $O(n \log n)$ approach compared to the naive $O(n^2)$ implementation, where n is the maximum degree among the polynomials.

SPIRAL

- SPIRAL is a code generation system that takes in high-level mathematical abstractions and synthesizes highly-optimized implementations.



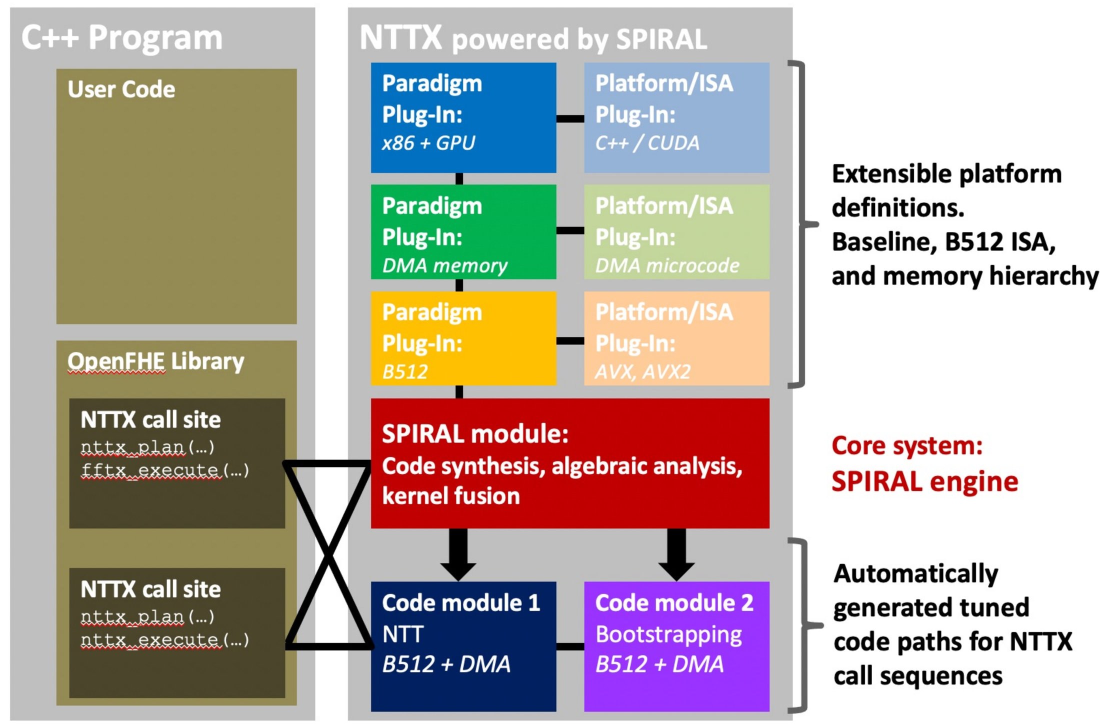
SPIRAL as an AI system.



Platform-Aware Formal Program Synthesis by SPIRAL.

SPIRAL NTTX Package

- Leveraging SPIRAL's capability of autonomous code generation and platform-based autotuning, we expand SPIRAL to the NTT domain.



NTTX powered by SPIRAL.

- Both the Korn-Lambiotte FFT algorithm and the Pease FFT algorithm are included as breakdown rules in SPIRAL to support general radix NTTs and simple parallelism.

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

NTTs of size r^k in SPIRAL's Operator Language.

- NTTX offers FFTW-style C/C++ API for FFTX-style code generation.

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

NTTX C/C++ API.

- As FHE requires large integers (e.g., 64-bit) for security, we focus on generating NTTs for multi-word integer data types on GPU.

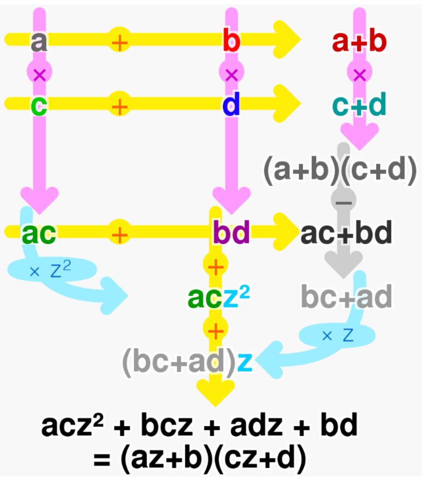
Multi-Word Arithmetic

- Using native integer data types, we implement multi-word/precision (MP) methods for three operations that NTT contains, namely (i) add, (ii) multiply, and (iii) modulo.
- The Barrett reduction algorithm is applied to compute modulo faster using multiplication, shifting, and subtraction rather than division.

$$a \bmod n = a - \lfloor as \rfloor n$$

Barrett reduction algorithm.

- We employ the Karatsuba algorithm to reduce the multiplication of two n-digit numbers to three multiplications of $n/2$-digit numbers.



Karatsuba algorithm.

CUDA NTT

- Constrained by the shared memory size of GPUs, the largest NTT for 64-bit integers that fits in one GPU thread block is of size 2,048 (i.e., 2,048-point 64-bit NTT).
- As the dataflow of NTT is sequential across stages, we allocate one thread block per NTT and compute batch NTTs using multiple thread blocks.

SPIRAL-Generated Multi-Word CUDA NTT

- Combining CUDA NTT with multi-word integer arithmetic, the SPIRAL NTTX package produces highly optimized multi-word CUDA NTT code.
- NTTs' correctness is verified against OpenFHE data.

```
// Host Code
void ntt2048mpcuda(uint64_t *Y, uint64_t *X,
    uint64_t modulus, uint64_t *twiddles, uint64_t mu) {
    dim3 b3(1024, 1, 1), g1(2, 1, 1);
    ker_code0<<<g1, b3>>>(X, Y, modulus, twiddles, mu);
}
```

```
// Kernel Code
__global__ void ker_code0(uint64_t *X, uint64_t *Y,
    uint64_t modulus, uint64_t *twiddles, uint64_t mu) {
    int a225, ...
    uint64_t s133, ...
    __shared__ uint64_t T1[2048];
    __shared__ uint64_t T2[2048];
    a225 = ((2048*blockIdx.x) + threadIdx.x);
    s133 = X[a225];
    s134 = _ModMulMP(twiddles[1], X[(a225 + 1024)], modulus, mu);
    a226 = (2*threadIdx.x);
    T2[a226] = _ModAddMP(s133, s134, modulus, mu);
    T2[(a226 + 1)] = _ModSubMP(s133, s134, modulus, mu);
    __syncthreads();
    ...
    s153 = T1[threadIdx.x];
    a245 = (threadIdx.x + 1024);
    s154 = _ModMulMP(twiddles[(1024 + (a245 % 1024))],
                     T1[a245], modulus, mu);
    a246 = ((2048*blockIdx.x) + (2*threadIdx.x));
    Y[a246] = _ModAddMP(s153, s154, modulus, mu);
    Y[(a246 + 1)] = _ModSubMP(s153, s154, modulus, mu);
    __syncthreads();
}
```

SPIRAL-generated radix-2 2,048-point MP CUDA NTT code, with a batch size of 2.

Results

- We benchmarked SPIRAL-generated batch NTTs' performance on Bridges-2 GPU nodes at Pittsburgh Supercomputing Center.
- The runtime of a single NTT is calculated as the overall kernel runtime of batch NTTs divided by the batch size.

Work	Device	n	Bit-Length	NTT [μs]
[2]	GTX Titan Black	1,024 2,048	24	2,160 2,060
[11]	Tesla V100	2,048	55	12.5
This Work	Tesla V100	1,024 2,048	60	0.24 0.56

Timings of a single SPIRAL-generated NTT on GPU and its comparison with other works.

- Although operating on integers of higher bit-lengths, SPIRAL-generated MP CUDA NTT achieves a **3,679x** speedup against [2] and a **22x** speedup against [11].

This material is based upon work funded and supported by Department of Defense under Contract No. HR0011-21-9-0003 with Carnegie Mellon University. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

[2] Pedro Alves and Diego Aranha. 2016. Efficient GPGPU implementation of the leveled fully homomorphic encryption scheme YASHE. Ph. D Dissertation. Master's thesis, Institute of Computing, University of Campinas, Brazil . . . .
[11] Özgün Özerk, Can Elgezen, Ahmet Can Mert, Erdinç Öztürk, and Erkay Savaş. 2022. Efficient number theoretic transform implementation on GPU for homomorphic encryption. The Journal of Supercomputing 78, 2 (2022), 2840–2872.