Generating Number Theoretic Transforms for Multi-Word Integer Data Types

Naifeng Zhang
naifengz@cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Franz Franchetti
franzf@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

1 Introduction

Fully Homomorphic Encryption (FHE) serves as a cryptographic approach that allows cloud platforms to manipulate encrypted data. Yet, a significant amount of computing power and time is required by FHE, where the bottleneck resides in polynomial multiplication. Of various implementations of polynomial multiplication, 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 [5] is a code generation system that takes in high-level mathematical abstractions and synthesizes highly-optimized implementations, which has outperformed domain experts across various platforms and kernels, especially in the domain of linear transforms such as the discrete Fourier transform (DFT). Leveraging SPIRAL's capability of autonomous code generation and platform-based autotuning, we expand SPIRAL to the NTT domain. As FHE requires large integers (e.g., 64-bit) for security, in this work, we focus on generating NTTs for multi-word integer data types on GPU.

2 NTTX

Mirroring the structure of FFTW [8] and FFTX [7], the NTTX package extends SPIRAL to generate NTT and batch NTTs [13]. As shown in Listing 1, 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);
```

Listing 1. NTTX C/C++ API.

Both the Korn-Lambiotte FFT algorithm [10] and the Pease FFT algorithm [12] are included as breakdown rules in SPIRAL to support general radix NTTs and simple parallelism. Using SPIRAL's Operator Language (OL) [6], NTTs of size r^k are represented as

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

2.1 CUDA NTT

To take advantage of the massive parallelism enabled by GPUs, we further expand the NTTX package to generate CUDA code based on the prior GPU support in SPIRAL's FFTX package. 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). Since 2,048-point NTT has 1,024 butterflies in each stage, and each thread block has 1,024 threads, we can perfectly parallelize a single stage of NTT within one thread block. As the dataflow of NTT is sequential across stages, we allocate one thread block per NTT and compute batch NTTs using multiple thread blocks.

Our implementation allows users to reuse 1,024 threads in a single stage through loop-based code for each stage, thereby generating NTTs of size larger than 2,048. However, the performance of NTT degrades as the larger but slower global memory is involved along with the shared memory.

2.2 Multi-Word Arithmetic

To support multi-word/precision (MP) integer arithmetic for NTTs, we implement MP methods for three operations that NTT contains, namely (i) add/sub, (ii) multiply, and (iii) modulo, using native integer data types. For addition and subtraction, multi-word carrying and borrowing are added to the code generator. We employ the Karatsuba algorithm [9] to reduce the multiplication of two n-digit numbers to three multiplications of $n/2$-digit numbers. The Barrett reduction [3] algorithm is applied to compute modulo faster using multiplication, shifting, and subtraction than division. In addition, we add new strength reduction rules to the SPIRAL internal compiler to reduce redundant variables and code.

Combining CUDA NTT with multi-word integer arithmetic, the SPIRAL NTTX package produces highly optimized MP CUDA NTT code, as displayed in Listing 2.

3 Results

We benchmarked SPIRAL-generated batch NTTs' performance on Bridges-2 GPU nodes at Pittsburgh Supercomputing Center [4], using one NVIDIA Tesla V100 SXM2 node with 32GB GPU memory and compute capability 7.0. The batch size is chosen as the maximum number of NTTs that fills up the entire GPU memory. The runtime of a single NTT is calculated as the overall kernel runtime (measured

Table 1. Timings of a single SPIRAL-generated NTT on GPU and its comparison with other works.

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

```
// 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();
}
// 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);
}
```

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

by nvprof) of batch NTTs divided by the batch size. NTTs' correctness is verified against OpenFHE [1] data.

To the best of our knowledge, there is limited work that implements small-size NTTs for large integers on GPU. Table 1 shows the performance comparison between SPIRAL-generated NTTs and other works using integer data types of different bit-lengths. 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].

Acknowledgments

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.

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 53–63.

[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 ....

[3] Paul Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 311–323.

[4] Shawn T Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A Nystrom. 2021. Bridges-2: a platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing*. 1–4.

[5] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[6] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. 2009. Operator language: A program generation framework for fast kernels. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 385–409.

[7] Franz Franchetti, Daniele G Spampinato, Anuva Kulkarni, Doru Thom Popovici, Tze Meng Low, Michael Franusich, Andrew Canning, Peter McCorquodale, Brian Van Straalen, and Phillip Colella. 2018. FFTX and SpectralPack: A first look. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*. IEEE, 18–27.

[8] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 3. IEEE, 1381–1384.

[9] Anatolii Alekseevich Karatsuba and Yu P Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.

[10] David G Korn and Jules J Lambiotte. 1979. Computing the fast Fourier transform on a vector computer. *Mathematics of computation* 33, 147 (1979), 977–992.

[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.

[12] Marshall C Pease. 1968. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM (JACM)* 15, 2 (1968), 252–264.

[13] Naifeng Zhang, Homer Gamil, Patrick Brinich, Benedict Reynwar, Ahmad Al Badawi, Negar Neda, Deepraj Soni, Kellie Canida, Yuriy Polyakov, et al. 2022. Towards Full-Stack Acceleration for Fully Homomorphic Encryption. (2022).