Accelerating High-Precision Number Theoretic Transforms using Intel AVX-512

Sophia Fu
Carnegie Mellon University
Pittsburgh, USA
syfu@andrew.cmu.edu

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

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

1 Introduction

Fully Homomorphic Encryption (FHE) allows various platforms to manipulate encrypted data offering ideal privacy protection. However, implementing FHE requires significant computing overhead, which can make it impractical in several applications. Researchers have previously addressed the computational overhead by developing accelerators specifically designed for FHE [6–8] or on server-level GPUs [5, 9]. While proven effective, we seek to accelerate FHE on CPUs, broadening the range of applications dramatically.

In this work, we choose to focus on Number Theoretic Transform (NTT), which is the computational bottleneck of multiple state-of-the-art FHE schemes that account for over 90% of FHE execution time in practice [3]. The NTT algorithm primarily depends on three integer modulo arithmetic operations; namely modular addition, modular subtraction, and modular multiplication. Specifically, we seek to optimize these operations on large multi-word integers of 128 bits and utilize Intel's Advanced Vector Extensions 512 (AVX-512) to parallelize our code. We then use these vectorized operations to accelerate any n-point NTT algorithm on a single CPU core.

2 Background

In this section we introduce necessary background for understanding AVX-512 and how it will be used to optimize multi-word arithmetic in a parallel NTT algorithm.

2.1 AVX-512

AVX-512 uses 512 bit vectors, which can be manipulated using specified SIMD (Single Instruction Multiple Data) instructions. More specifically, these vectors can handle eight 64-bit integers, allowing us to perform eight modular operations in parallel. While it is possible to directly insert assembly instructions into the source code, Intel also supports intrinsics which allows SIMD instructions to be treated as C functions. Thus, by utilizing intrinsics, we are able to incorporate AVX-512 directly into our code.

2.2 Multi-Word Arithmetic

To implement NTT, we first implement our three integer modulo arithmetic operations: modular addition, subtraction, and multiplication. We use simple arithmetic and bitwise operations in C to implement large integer arithmetic, utilizing both 128-bit and 64-bit representations. These scalar operations seek to primarily optimize the modulus operation, especially modular multiplication. To do so, we utilize the Karatsuba algorithm along with the Barrett Reduction algorithm chosen by prior literature [8, 9]. For example given $x * y$ where $x = (10a+b)$ and $y = (10c+d)$, we can use smaller bit widths and expand the equation to $100ac + 10(ad + bc) + bd$. Using smaller bit widths allows us to perform multiplication more efficiently and addresses overflow.

2.3 Number Theoretic Transform

NTT is an established algorithm to perform Discrete Fourier Transform (DFT) on finite fields within $O(n \log n)$ time. Given an input $x = x_0, ..., x_{n-1}$, such that $x_j \in [0, m)$, where m is some modulus, the outputs $y = y_0, ..., y_{n-1}$ are calculated using the equation:

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} \mod m \tag{1}$$

in which ω_n is the n^{th} primitive root of unity.

The NTT algorithm provides a more efficient implementation. Instead, for a n-point NTT we have $\log_2 n$ stages, where each stage has $\frac{n}{2}$ butterfly operations. Each butterfly operation consists of a modular multiplication, an addition and a subtraction. We choose to use the Pease algorithm to maximize parallelism. In particular, the butterflies in each stage have no dependencies on each other, allowing us to calculate several butterflies in parallel.

3 Related Work

To date, SIMD intrinsics have primarily been used to parallelize modular arithmetic on integers up to 64 bits for NTT calculations. The Mathemagix library provides Intel AVX2 implementations of modular arithmetic operations on 32 and 64-bit integers [4]. Intel HEXL [1] built upon these established algorithms, using Intel AVX512-IFMA52 to continue to accelerate finite field arithmetic on 64-bit integers.

In this paper, we focus on expanding upon these kernels established in Intel HEXL to larger integers of 128 bits, utilizing AVX-512. Currently, for operations on 128-bit integers or larger, the GNU Multiple Precision Arithmetic Library (GMP) is regarded as the state of the art. Within GMP, multi-digit integers are represented using an array of limbs, with each limb storing a part of the multi-precision number that fits in a single machine word. Building upon these ideas, we seek to represent 128 bit integers as two 64-bit numbers and use AVX-512 intrinsics to parallelize several operations at once.

4 Proposed Approach

We first implement modular addition, subtraction, and multiplication using AVX-512 instructions, to offer maximum parallelism. Using these operations, we then parallelize butterfly operations in the NTT algorithm.

```
// Parallel modular subtraction on 8 inputs, returning results in ch and cl
uint128_t submod128(__m512i* ch, __m512i* cl, __m512i ah, ..., __m512i ml) {
    t30 = _mm512_sub_epi64(al, bl);
    c1_m = _mm512_cmp_epu64_mask(al, bl, _MM_CMPINT_LT);
    t28 = _mm512_mask_add_epi64(bh, c1_m, bh, one);
    t29 = _mm512_sub_epi64(ah, t28);
    i28_m = _mm512_cmp_epu64_mask(ah, t28, _MM_CMPINT_LT);
    ...
    d3 = _mm512_add_epi64(d2, mh);
    *ch = _mm512_mask_blend_epi64(i28_m, t29, d3);
    *cl = _mm512_mask_blend_epi64(i28_m, t30, d1);
}
```

Listing 1: Modular subtraction C code using AVX-512.

4.1 SIMD Vectorized Modular Arithmetic

For each of the three modular operations, we first take a scalar algorithm utilizing only 64-bit integers and translate each arithmetic and bitwise operation to their AVX-512 equivalent. This allows us to translate our code to handle eight operations in parallel.

Since AVX-512 can only support operations on 64-bit integers, each 128 bit input is split and stored in separate vectors. Thus, instead of taking in a single 128 bit integer `a` at a time, two 512 bit vectors are passed in as `ah` and `al`, representing eight 128 bit integers. `ah` contains the upper 64 bits of each of the eight 128 bit integers, and `al` contains the lower 64 bits. In Figure 1, we illustrate our strategy of implementing a modular subtraction using AVX-512 which corresponds to the code is shown in Listing 1.

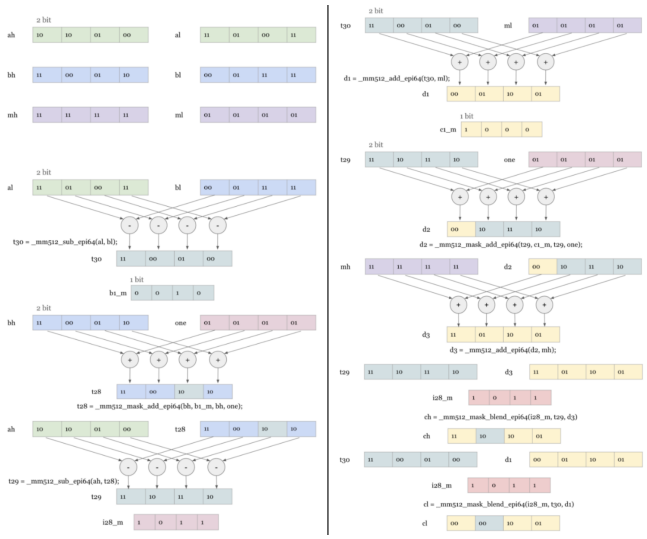

Figure 1: Overview of modular subtraction using AVX-512.

4.2 SIMD Vectorized NTT

Using our established parallel operations, we then integrate them into the NTT algorithm. To do so, we must organize our data into vector formats.

Twiddle Factor Vector Generation. A certain set of twiddle factors are used for the butterflies in a each stage. These factors are typically calculated prior to the NTT and stored within an array.

We want to load twiddle factors into arrays of vectors. For an n-point NTT, with $\log_2 n$ stages and $\frac{n}{2}$ butterflies in each stage, there are $\frac{n*\log_2 n}{2}$ twiddle factors to load. Since the upper 64-bits and lower 64-bits of each factor are stored separately, we keep two arrays of vectors, `twdh` and `twdl`, each with length $\frac{n*\log_2 n}{16}$.

To load each vector, we first consider the initial three stages. Each of the vectors in these stages holds eight non distinct integers following a repeated pattern, allowing us to easily preload these three vectors. These vectors are then loaded into the array such that there are $\frac{n}{16}$ vectors corresponding to each stage.

Beyond the first three stages, we load our vectors sequentially. At any given stage i, there are 2^{i-3} distinct vectors. These distinct vectors are loaded into the array and copied for the remaining $\frac{n}{16} - 2^{i-3}$ vectors for the given stage.

Generating Input Vectors. Given an n-point NTT, the inputted array consists of $2n$ 64-bit integers, where each pair of 64-bit integers represents one 128 bit integer. We first load these values into an array of $\frac{n}{4}$ vectors using the `_mm512_load_epi64` instruction. Then, we use `_mm512_unpacklo_epi64` and `_mm512_unpackhi_epi64` to shuffle consecutive pairs vectors so that the two resultant vectors hold the higher 64 bits of each input or the lower 64 bits respectively. These vectors are stored into arrays `xh` and `xl` to be used as our inputs to each butterfly in the first stage.

Butterfly. Once the inputs are loaded, we can perform the butterfly operations. Each butterfly operation involves modular multiplication, addition and subtraction. These butterflies can be performed in parallel using our established modular operations. Thus, for a n-point NTT with $\frac{n}{2}$ butterflies, we only need to perform $\frac{n}{16}$ sets of operations.

Shuffling. The results from each previous stage must be permuted for each decomposition. As shown in Figure 2, for the first stage we can use `_mm512_unpacklo_epi64` and `_mm512_unpackhi_epi64` as the results are ordered differently. For each of the other stages, we use `_mm512_permutex2var_epi64`, permuting the vectors `ah` and `sh` together, along with `al` and `sl`.

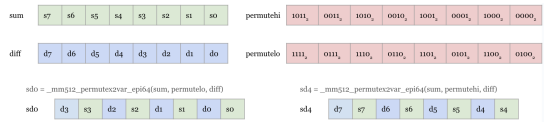

Figure 2: Shuffling between stages using AVX-512 permute instructions.

5 Results

For performance evaluation, we benchmarked various sizes of NTT on FASTER nodes at Texas A&M University [2]. We used the Intel oneAPI DPC++/C++ Compiler 2023.2.0, running the code using a single core on the Intel Xeon 8352Y (Ice Lake) processor. The runtime of a single NTT was found by averaging the last 10 iterations over 30 total runs, allowing the cache to warm and stabilize. We excluded the data preprocessing time for our implementation and other baselines.

We implemented various sizes of NTT using the GMP library as a baseline. Furthermore, we used the SPIRAL NTTX package [9] to generate efficient scalar C NTT implementations and compared against them. In our implementation, one butterfly function (that

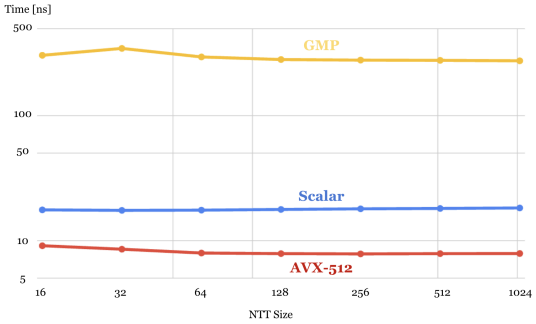

Figure 3: Runtime per butterfly for various NTT sizes.

computes 8 butterflies in parallel) consists of 332 lines of code and takes 176 cycles when measured by LLVM Machine Code Analyzer. From Figure 3, we see that compared to GMP, the vectorized code offers 36 times speedup on average across all NTT sizes. Additionally, compared to the SPIRAL-generated scalar code, we can still offer around 2.2 times speedup on average.

Acknowledgments

This work used FASTER at Texas A&M University through allocation CIS230287 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

[1] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. 2021. Intel HEXL: accelerating homomorphic encryption with Intel AVX512-IFMA52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 57–62.

[2] Timothy J Boerner, Stephen Deems, Thomas R Furlani, Shelley L Knuth, and John Towns. 2023. ACCESS: Advancing Innovation: NSF's Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support. In *Practice and Experience in Advanced Research Computing*. 173–176.

[3] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[4] Joris Van Der Hoeven, Grégoire Lecerf, and Guillaume Quintin. 2016. Modular SIMD arithmetic in Mathemagix. *ACM Transactions on Mathematical Software (TOMS)* 43, 1 (2016), 1–37.

[5] 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.

[6] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 238–252.

[7] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[8] Deepraj Soni, Negar Neda, Naifeng Zhang, Benedict Reynwar, Homer Gamil, Benjamin Heyman, Mohammed Nabeel, Ahmad Al Badawi, Yuriy Polyakov, Kellie Canida, et al. 2023. Rpu: The ring processing unit. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 272–282.

[9] Naifeng Zhang and Franz Franchetti. 2023. Generating number theoretic transforms for multi-word integer data types. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.