EFFICIENT IMPLEMENTATION OF THE NUMBER THEORETIC TRANSFORM

Rudolf Loretan, Artur Melo, Rijad Nuridini, Philippe Panhaleux

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The Number Theoretic Transform (NTT) has many practical applications. Computing it as efficiently as possible is therefore crucial. In this paper, we present two fast implementations of the number theoretic transform based on a radix-2 iterative and a radix-4 recursive algorithm in C++. The optimizations include various scalar tricks and use of Intel vector intrinsics. We explore the effects of implementing the transform iteratively or recursively, and varying the radix. Our best implementation achieves on average a 16.5x performance increase over our scalar baseline implementation and 7.3x over a selected, publicly available NTT library. In a final step we conduct a roofline analysis to show that the implementations are compute bound.

1. INTRODUCTION

Motivation. The Number Theoretic Transform (NTT) is a variant of the Discrete Fourier Transform (DFT) operating on a finite field of integers. NTT allows efficient cyclic and negacyclic convolutions which have many applications in computer arithmetic, e.g., for multiplying large integers (Schönhage–Strassen algorithm[1]) and large degree polynomials. Further, it finds application in numerous cryptographic schemes and image processing[2, 3].

Contribution. Our goal is to experiment with different NTT algorithms in a first step. We subsequently optimize the most promising implementations as much as possible. This means achieving the maximum performance for our specific machine and micro-architecture while maintaining support for reasonably large numbers and input sizes. Finally, we analyze their performance and memory usage.

Related work. FFTs have been the topic of many publications and a wealth of different algorithms exist. These include $O(n \log n)$ algorithms for any input size n, as well as for various computing platforms. The most commonly used DFT by far is for powers of two input sizes n, partly because these sizes permit the most efficient algorithms[4]. Generally, a lot of this research is applicable to NTT but there are some substantial differences that can introduce other obstacles. For example, the large number of modulo operations likely leads to another bottleneck. On the other hand, since NTT does not operate on complex numbers, it should require less data movement than DFT.

While there exist a number of high performance DFT libraries (like FFTW[5] and libraries generated by Spiral[6]), the amount of fast NTT libraries is very limited. Regarding actual NTT optimization, there are multiple papers that propose arithmetic improvements for the modular reduction algorithm inside NTT[2, 7, 8]. These, however, usually come at a cost. Reducing the number of modulo operations generally introduces restrictions on input length and size. To our knowledge, there are only very few publications that incorporate SIMD (Single Instruction Multiple Data) into NTT computation[8, 9, 10].

2. BACKGROUND ON THE ALGORITHM

In this section we formally define the number theoretic transform, introduce the algorithms we use, and present the performance cost analysis.

2.1. The Number Theoretic Transform

NTT is in its essence a DFT where the complex numbers are replaced by numbers over the ring of integers modulo a prime m [11]. For an NTT of length n we need a primitve n^{th} root of unity. A number r is called an n^{th} root of unity if $r^n = 1 \mod m$. It is called a primitve n^{th} root if $r^k \neq 1 \mod m, \ \forall k < n$. Given n integer inputs $x_0, \ldots, x_{n-1}$, the NTT is defined as

$$Y_k = \sum_{i=0}^{n-1} x_i \cdot r^{k \cdot i} \tag{1}$$

In the same manner as FFT can be constructed from DFT, a "fast number theoretic transform" can be constructed from NTT[11].

Fast NTT. In this paper, we present an adaptation of the most common FFT algorithm, the Cooley–Tukey FFT algorithm (CT[12]). It reduces the overall time complexity of NTT to $\mathcal{O}(n \log n)$ using divide and conquer[13].

Recursive or Iterative. CT divides the given problem into sub-problems and solves the original problem by using the

result of the sub-problems. It can be implemented both recursively and iteratively. Its recursive implementation can be defined the following: Given integer vector of size N,

$$NTT_N = (NTT_m \otimes I_n) D_{m,n} (I_m \otimes NTT_n) L^{mn}_m, \quad (2)$$

where $N = km$. m is called the radix and $D_{m,n}$ is a diagonal matrix containing the twiddle factors:

$$D_{m,n} = \bigotimes_{j=0}^{m-1} \text{diag}(r^0_{mn}, r^1_{mn}, ..., r^{n-1}_{mn})^j \quad (3)$$

L^{km}_k is the stride permutation matrix. It permutes the input vector x of length km as $im + j \mapsto jk + i$, $0 \leq i < k, 0 \leq j < m$.
For the iterative CT we have the following equation:

$$NTT_N = (\prod_{i=1}^{k} (I_{2^{i-1}} \otimes NTT_2 \otimes I_{\frac{N}{2^i}}) D'_{N,i}) R_N, \quad (4)$$

where $D'_{N,i}$ are diagonal matrices and R_N is the bit-reversal permutation. Literature[14] suggests that recursive CT generally has better locality and should be preferred. Especially when targeting a machine with a memory hierarchy, starting the optimization with iterative radix-2 FFT is suboptimal since it requires $log_2(N)$ many sweeps through the input data which results in poor cache locality.

Vector-Radix Algorithms. The radix of a FFT/NTT algorithm is the number of sub-NTTs each stage is split into, and also the number of vector elements each base case takes as input. A radix-2 base FFT takes two elements as input and has two outputs. To obtain a radix-4 FFT algorithm, we essentially merge two sequential radix-2 stages into one. A radix-4 base FFT therefore has 4 inputs and outputs. These FFT/NTTs are then combined into larger ones. This process is called a butterfly.

Selected Radix-4 Algorithm. The radix-4 base algorithm we will implement, optimize and discuss in the following sections is in essence the NTT equivalent of the radix-4 FFT described in "How To Write Fast Numerical Code: A Small Introduction"[14]. Notice that this implementation fuses the first two and last two matrices in equation (2) to obtain a 2-stage algorithm with improved locality.

Comparison FFT and NTT. In his 1988 publication[15], David H. Bailey argues that NTTs are inherently slower than any normal FFT but require less space. Morever, they do not have any round-off errors as NTT uses integers instead of doubles. This is the reason why NTT is suitable for multiplying large integers.

2.2. Cost Analysis

To measure the cost of the radix-2 and radix-4 algorithm, all integer additions, multiplications, divisions and modulo operations were counted. We did not consider loop or index calculations, as those are not directly related to the transform itself (similar reasoning as in [16]). The accuracy of both cost functions was verified by embedding simple counters in the code.

Radix-2 Iterative.

$$C_{\text{radix2-iter}}(n) = \frac{n \log_2 n}{2} (1 C_{\text{mul}} + 1 C_{\text{mod}} + 6 C_{\text{add}}) \quad (5)$$

Radix-4 Recursive.

$$C_{\text{twiddle}} = 25 C_{\text{add}} + 6 C_{\text{mul}} + 5 C_{\text{mod}} \quad (6)$$

$$C_{\text{base}} = 24 C_{\text{add}} + 1 C_{\text{mul}} + 1 C_{\text{mod}} \quad (7)$$

$$C_{\text{radix4-rec}}(n) = \begin{cases} \frac{n}{4}((\log_4 n \text{ - } 1) C_{\text{twiddle}} + C_{\text{base}} + 1 C_{\text{div}}) & \text{if } 4|n \\ 2 C_{\text{radix4-rec}}(\frac{n}{2}) + \frac{n}{2} C_{\text{step}} & \text{o/w} \end{cases} \quad (8)$$

where $C_{\text{step}} = 6 C_{\text{add}} + 1 C_{\text{mul}} + 1 C_{\text{mod}}$. We split the cost function of the radix-4 recursive algorithm depending on whether the input size is a power of 4. If it is not, we must compute one additional radix-2 twiddle step, which is represented by $\frac{n}{2} C_{\text{step}}$. For both algorithms, we obtained a complexity of $\mathcal{O}(n \log n)$, which is consistent with the theoretic complexity of the Cooley-Tukey FFT.

3. OUR PROPOSED METHOD

We implemented both radix-2 recursive and iterative as well as the radix-4 recursive variants of the Cooley-Tukey algorithm. The radix-2 recursive version displayed notably worse performance than the other options, so we did not pursue it further. As the most expensive instruction is the modulo, we replaced it with overflow checks (2) in modular additions and subtractions. Where we could not replace the modulo operations with overflow checks, for example in modular multiplication, we used floored division. We also applied scalar replacement, loop unrolling, and inlining to further improve performance. Finally, we used Intel vector intrinsics to make use of SIMD operations.

3.1. General Optimizations

Strategies mentioned in this section find application in all our implementations.

Precomputation. As twiddle factors only change with different moduli or input sizes but not with the input vector, one can precompute those weights. For a large number of transforms of the same size and modulus, this computation

can be amortized. This calculation as well as the initialization of most other variables and vectors were removed from the transform and put into a separate `init` function.

Scalar Replacement and Loop Unrolling. Even though the compiler likely takes care of scalar replacement, we generally followed the high-performance best practice of first loading values from memory, then doing the computation, and storing the values at the very end. Unrolling certain loops and inlining the function calls they contain also gave a small speedup. These tricks are already present in our baseline implementations.

Overflow Checks. Instead of implementing a regular modular addition such as in algorithm (1), we decided to replace it with if statements checking whether the value would overflow, as shown in algorithm (2). This optimization is also present in our baseline implementations.

Algorithm 1: Standard addition

Input : int a,b, prime n
Output: x

1 $x \longleftarrow a + b \mod n$;

Algorithm 2: Overflow checked addition

Input : int a,b, prime n
Output: x

1 **if** $a + b > n$ **then**
2 $\quad x \longleftarrow a + b - n$
3 **else**
4 $\quad x \longleftarrow a + b$
5 **end**

The same logic was applied for modular subtraction but the algorithm is omitted.

Floored Division. Instead of the regular modulo operation, we used floored division as described by Donald Knuth[17]: $a \mod n = a - n \cdot \lfloor \frac{a}{n} \rfloor$. While testing, we found that this method has similar performance to the standard definition of `mod` in `C`. Since SSE2 intrinsics do not include any modulo operation, this trick enabled us to implement it anyway.

Precomputed Division. Having replaced the modulo operator with a floored division, we noticed that the $\frac{1}{n}$ division was identical in every modular multiplication. By precomputing this value, we only need to do one costly division and can do a multiplication with the result in all other occurrences.

FMA Operation. We tried to replace the multiply-subtract of the floored division with a fused `fnmadd` operation. This lead to a negligible performance improvement (<1%), so we will omit this attempt from our analysis in section (4).

3.2. Radix-2 Iterative

Scalar Optimizations. For the radix-2 iterative variant, we implemented overflow checks (2) and floored divisions (5). We also tried replacing the division operations with shifts, as our code has only divisions by powers of two. This did not lead to any performance improvements (see (4.2)).

```
for(int len=2, half=1; len<=n; len<<=1, half<<=1)
{
    int i = log2(len) - 1;
    ull *T = precomp_radix2_iterative[i];
    for(int i = 0; i < n; i += len)
    {
        for(int j = 0; j < half; j++)
        {
            ull u = outvec[i + j];
            ull v = (T[j]*outvec[i+j+half])%mod;
            MOD_ADD(outvec[i+j], u, v, mod);
            MOD_SUB(outvec[i+j+half], u, v, mod);
        }
    }
}
```

Listing 1. Radix-2 main loop

Looking at the main loop of radix-2 in listing (1), we can see that unrolling the inner loops leads to potential out of bounds errors. To bypass this, we extracted the loop-body for the initial iteration with small values `len` and `half`.

As an example, the first extraction for `len=2`, `half=1` is shown in listing (2). We see that we can remove the outer and the inner-most loop of the original function. Additionally, we replaced all index calculations that included `j`, `len`, and `half`. This was done for `len=2` and `len=4`. For larger `len`, loop unrolling was viable and we did not run into out of bounds errors. The code below also includes the aforementioned floored division (5) in the `MOD_2` method call. `MOD_ADD` and `MOD_SUB` are the overflow checked methods.

```
    // len = 2, half = 1
    if (n < 2) {
        return;
    }
    ull *T = precomp_radix2_iterative[0];
    ull T_0 = T[0];
    for (int i = 0; i < n; i += 2) {
        ull u = outvec[i];
        ull v = MOD_2(T_0 * outvec[i + 1]);
        MOD_ADD(outvec[i    ], u, v, mod);
        MOD_SUB(outvec[i + 1], u, v, mod);
    }
```

Listing 2. Radix-2 main loop unrolled by factor 2

Vector Optimizations. Most of the scalar operations can be modelled with a combination of SSE2 and AVX intrinsics. Since we did not have access to a CPU with AVX-512 support (which is required to correctly multiply vectors of 32 bit numbers) we decided to use packed double vectors for

modular multiplication and convert our integers back and forth. We mainly used SSE2 as AVX/AVX2 do not have some key integer operations that are required (certain comparisons, loads and stores to be specific). We had to use AVX intrinsics for the modular multiplication because the floor operation is not available for 256-bit operands. We also tested our code with solely packed-double intrinsics but it was less efficient than using SSE2 integer operations.

3.3. Radix-4 Recursive

We now present the optimizations we implemented for the radix-4 recursive algorithm.

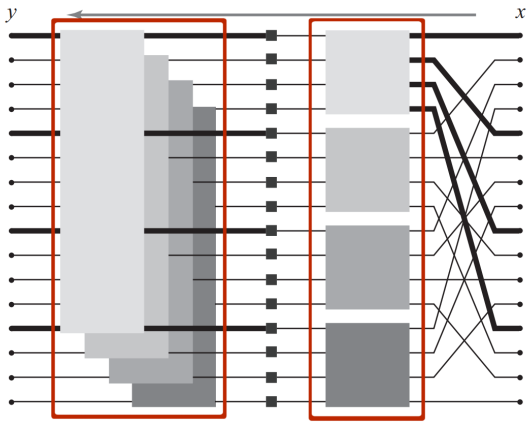

Fig. 1. Radix-4 recursive algorithm ($n = 16$) vectorization scheme, with twiddle functions on the left and the base case functions on the right

Scalar Optimizations. As for radix-2, we started by replacing all modular additions and subtractions with overflow checks (2). We also replaced modular multiplications using the precomputed floored division mentioned previously (5). Since the radix-4 code was already optimized to some extent, we could not get much further using only scalar tricks.

Vector Optimizations. Profiling our code at this point revealed that during execution, most time by far is spent running the twiddle function. The second hotspot turned out to be the base case function (its code is illustrated as an example in listing (3)). The overhead imposed by the recursion and helper functions was comparatively small. We hence decided to vectorize these two functions. However, it quickly became apparent that they cannot be vectorized on their own as the operations are all dependant on one another. What is possible, though, is to execute four twiddle resp. base functions simultaneously, as shown in red in figure (1). Additionally, there needs to be a transpose operation between the base and twiddle function for this to work. For the same reasons as in (3.2), we used SSE2 instead of AVX-2 intrinsics. Furthermore, 4 integers fit precisely into one SSE2 vector which really suits the radix-4 algorithm.

```
void NTT4_base(unsigned *Y, unsigned *X, ull s)
{
    ull X0 = X[0 * s];
    ull X1 = X[1 * s];
    ull X2 = X[2 * s];
    ull X3 = X[3 * s];
    ull t0, t1, t2, t3;

    MOD_ADD(t0, X0, X2, mod);
    MOD_SUB(t1, X0, X2, mod);
    MOD_ADD(t2, X1, X3, mod);
    MOD_SUB(t3, X1, X3, mod);

    t3 = MOD(baseroot * t3);

    MOD_ADD(Y[0], t0, t2, mod);
    MOD_ADD(Y[1], t1, t3, mod);
    MOD_SUB(Y[2], t0, t2, mod);
    MOD_SUB(Y[3], t1, t3, mod);
}
```

Listing 3. Radix-4 base function

Cache Optimizations. It can be seen from equations (2) and (4) that in NTT the vectors are accessed with a stride such that there is no spatial locality. Modern processors usually use some variant of a least recently used (LRU) replacement policy and the data is loaded in cache-line sized blocks. This means that with large strides additional capacity misses are likely to occur. The pseudocode in algorithm (3) shows how the input can be reordered in order to reduce the number of cache misses. However, while benchmarking, we generally noticed a performance degradation and there was more overhead than benefit. Only the largest test cases showed a visible improvement. It would be very interesting to see the effects of implementing the reordering with lookup tables. This could make the loop much faster. Unfortunately this was not tried due to time constraints.

Algorithm 3: Reorder vector for better spatial locality

Input : vector X, size n
Output: vector Y

```
s ← n/16;
for i ← 0 to s do
    for j ← 0 to 16 do
        Y[16 * i + j] ← X[i + j * s];
    end
end
```

4. EXPERIMENTAL RESULTS

In this section we describe our experimental setup, analyze performance and memory usage of the implementations, and discuss the results.

4.1. Methodology

Validation. We generated random test cases of various sizes to validate all implementations. These transforms were done using the `ntt` function in the python sympy library[18].

Experimental Environment. The same machine was used for all experiments. It is running Ubuntu 18.04 with kernel 4.15.0-101-generic on an Intel Core i7 7700 Skylake CPU. It has 32KB of L1, 256KB of L2 and 8MB of L3 cache. Intel Turbo Boost is switched off to achieve a constant CPU frequency of 3.6 GHz. All code was compiled using `g++` version 7.5.0 with flags `-O3 -march=native`. We tried other flags and compilers, but got slightly worse results.

Tools. The RDTSC instruction was used to access the x86 Time Stamp Counter (TSC) for accurate performance measurements. Moreover, we used Intel VTune and Intel Advisor [19] to find hotspots and bottlenecks.

Performance Measurements. Our benchmark suite allowed us to specify code that is not timed and is executed before the measured section (we call this the `init` function). After warming up the CPU the implementation is timed repeatedly. In the end the average amount of cycles required to run the implementation is stored.

Benchmark Alternative. To measure our improvements we used a library called Finite Transform Library (FTL[20]). Only later did we discover that it is not really an optimized high-performance library.

4.2. Performance Results

In all following performance plots the x-axis denotes the input vector size and the y-axis shows the corresponding performance measured in integer operations (intops) per cycle. Note that the blue curve in all succeeding performance figures represents our benchmark alternative FTL. Further, the vertical lines in all plots indicate where the working sets reach L1, L2 and L3 cache sizes. The working set generally consists of an input and an output vector each of size n. However, it is very hard to determine the exact size of the working set as it is different for recursive and iterative algorithms. The cache bounds may therefore vary and should be treated with caution.

Radix-2 Scalar Optimizations. In figure (2) we present all the scalar optimizations. The orange line shows our base radix-2 iterative version. This version performs similarly to the benchmark alternative although it stays constant for larger input sizes in contrast to FTL, which starts to decline. The green line represents the version where we replaced divisions with the corresponding shift operation, as we can see this optimization did not give us any performance improvements. This is due to `g++` replacing division by powers of two with shifts even with no optimizations enabled. Replacing modulo operations with floored division (5) did improve

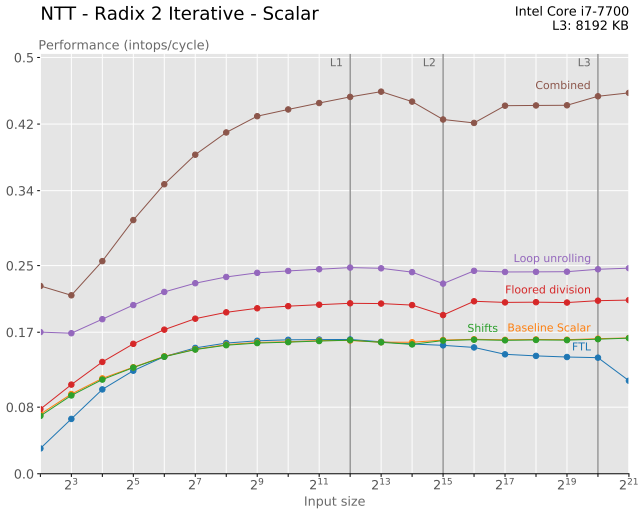

Fig. 2. Radix-2 scalar optimizations, shows the speed up for overflow checks (2), floored division, (5) and no improvement for shifts instead of divs. Speedup of 3.5x over FTL

our performance, we can see this in the red line on the plot. The purple line contains the unrolling as presented in (2), which, to our surprise, also lead to a performance increase. When combining loop unrolling and floored division we get the brown line. It offers a speed-up of roughly 3.5x compared to the benchmark alternative and reaches roughly 11% of our peak scalar int performance.

Radix-2 Vector Optimizations. The green line in figure (3) shows our best scalar implementation for comparison. The orange curve is the baseline vectorized version without floored division (5), but with the changes mentioned in (2). Finally the red line includes both overflow checks (2) and floored division (5). This corresponds to roughly 7% of the theoretical peak vector int performance.

Radix-4 Scalar Optimizations. Figure (4) demonstrates the performance of our radix-4 scalar improvements. The baseline version (orange) corresponds to the radix-4 algorithm described in (2.1). Thus it already contains some principles of performance-conscious programming, such as not using complicated structs, not using dynamically generated data structures, and other procedures fundamental to writing fast code. The green curve is obtained by precomputing as many steps as possible and moving this code into the initialization function. This includes all twiddle factors and other intermediate values required. The fastest implementation (red curve) shows the speedup gained by replacing the modulo operations with a precomputed floored division. The best implementation offers on average a 4x performance boost over the baseline version we started from. It reaches roughly 13.75% of our peak scalar int performance.

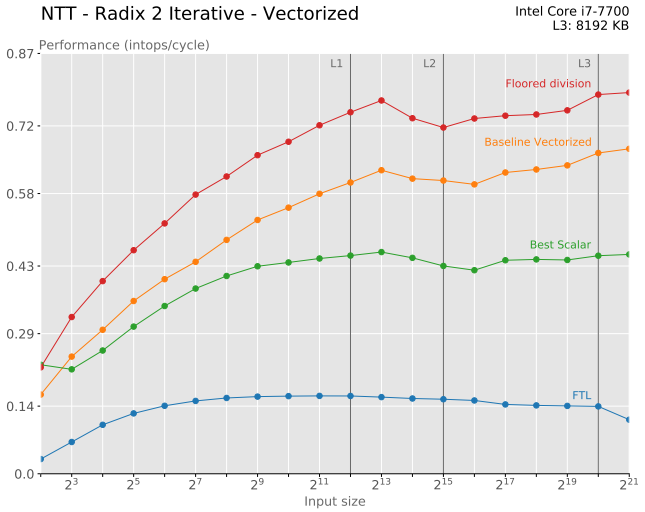

Fig. 3. Radix-2 vector optimizations, shows the improvements of using SIMD, and when adding overflow checks (2), loop unrolling, as well as for floored division

Radix-4 Vector Optimizations. The orange line in figure (5) shows our first vectorized implementation which led to an interesting sawtooth pattern. The reason being that radix-4 naturally works for powers of four but for the remaining powers of two performs a radix-2 step first. So this step had to be vectorized as well, which led to the red curve. The purple line shows the final performance improvements obtained by integrating the finishing optimizations of the scalar code. In the end an average speedup of 3.5x compared to the best

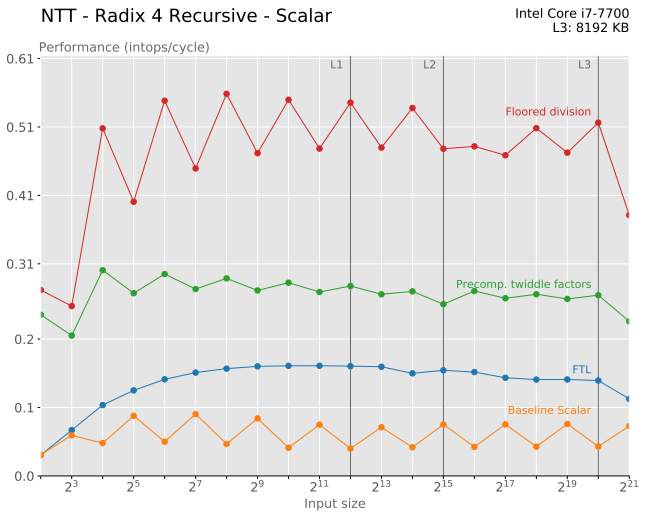

Fig. 4. Radix-4 scalar optimizations, shows basic performance aware implementation, precomputation of twiddle factors and replacing modulo with floored division

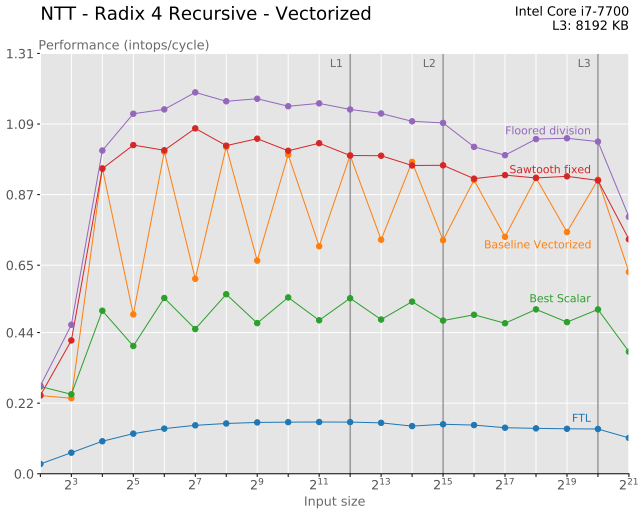

Fig. 5. Radix-4 vector optimizations, shows performance gains of vectorizing base and twiddle functions (red), and final version using all tricks (purple)

scalar implementation, 7.3x over FTL, and 16.5x over the scalar baseline was achieved. The code runs at roughly 10% of the theoretical peak vector int performance.

Overview. Figure (6) shows an interesting trend: radix-4 turns out to be noticeably faster than radix-2. This can be explained in that radix-4 is more adapted to vectorization, since there are always at least 4 elements that can be comfortably put into a vector. We can also see that by comparing radix-4 vectorized with radix-4 scalar. Radix-2 vectorized still has potential for improvements on lower input sizes. But we refrained from vectorizing the smaller sizes, as some fundamental restructuring of the algorithm would be required to eliminate the out of bounds errors. The final trend we can see is that the performance curves tend to rise at the beginning and flatten at some point after hitting L1 cache size. This might be due to the impact of uncounted index operations being larger for smaller test sizes, but we are not certain.

Comparing Radix-2 and Radix-4. Radix-2 scalar has about 10% less operations compared to radix-4 based on our cost analysis in section (2.2). Regardless, our radix-4 implementation displays a better performance and runtime. A bigger difference can be seen when both implementations are vectorized. In that case we have a 50% performance difference. We argue that this boils down to the different structure of both algorithms. Since we implemented radix-2 iteratively and radix-4 recursively we cannot say for sure which of the two decisions lead to the significant performance difference, but we are certain that radix-4 is better suited for using SIMD.

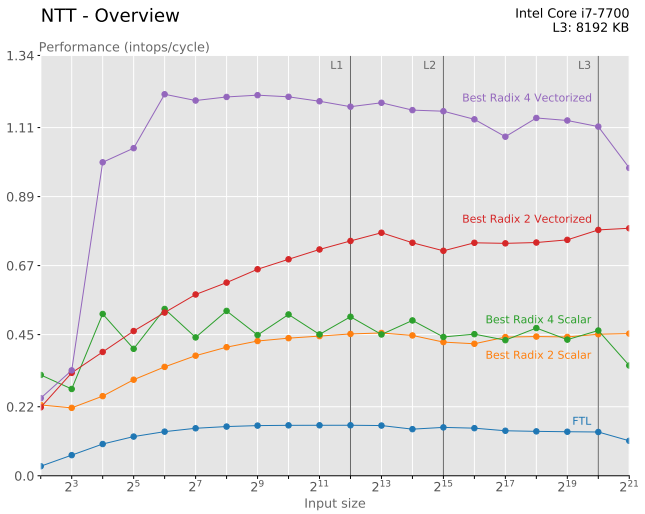

Fig. 6. Overview: Comparison of our best implementations, radix-2 and radix-4, scalar and vectorized

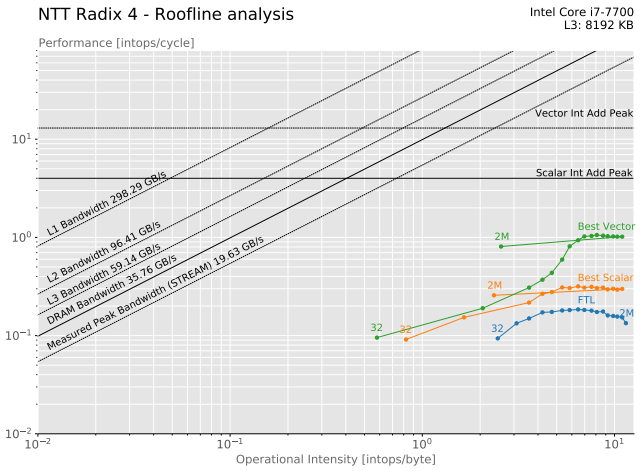

Fig. 7. Roofline plot of the best implementations, cold cache measurements, all three are compute bound

4.3. Roofline Analysis

The roofline model is a powerful performance analysis tool to visualize inherent hardware limitations and room for improvement. It integrates performance, memory bandwidth, and locality into one single comprehensive plot. The horizontal axis represents the operational intensity of the computation, defined as the ratio of number of operations performed per byte of data moved between main memory and the processor. The lower the operational intensity, the more constrained is the computation by the memory bandwidth. As the operational intensity increases, the upper bound on achievable performance steadily increases up to the point where the theoretical peak performance of the processor will limit performance.[21, 22]

The most challenging part is determining data movement since it depends on cache size, coding decisions, and hardware. We used PAPI[23] to count the number of L3 misses in our code. The greatest difficulty was that the hardware prefetcher moves data into the cache that is not counted. We were able to show this exact behaviour using small programs that access elements of a large array in either some predictable or in random order. However, NTT mainly requires one input and one output vector, and the amount of space required for intermediate values is relatively small. Additionally almost all of our input sizes (up to 2^{19}) fit comfortably into L3 cache. Therefore we used the maximum of memory movement registered by PAPI, and the input/output vectors as our data movement. This results in a solid lower bound for memory movement, which in turn is an upper bound for operational intensity. Figure (7) shows that our NTT implementations as well as the benchmark alternative tend to be compute bound for input sizes 2^5 to 2^{21}.

However, there is an interesting trend visible for the largest test cases : The number of L3 misses increases significantly and therefore operational intensity drops. This can be explained by the large strides in which the vectors elements are accessed. As the vector size approaches a multiple of the L3 cache size, the number of capacity misses rises. This could be countered by reordering the data at the beginning, but it would mainly benefit the largest input sizes which are rarely used in practice. Simultaneously the extra overhead decreased performance for the other test cases, which is why we did not pursue this further.

5. CONCLUSIONS

In this paper we compared and optimized multiple variants of the number theoretic transform. Even though the radix-4 baseline code described in [14] was already fast, we achieved an average performance gain of 16.5x with our final version. During the course of optimization the first major improvement came from maximizing precomputation. Subsequently we significantly reduced the inherent bottleneck imposed by the modulo operations. Finally we focused on the the functions that are called most often: NTT-base and NTT-twiddle. In spite of the very limited amount of integer intrinsics in vector extensions prior to AVX-512 we managed to fully vectorize both functions. The roofline analysis shows that our implementations are clearly compute bound for these input sizes. For radix-2 we could not quite reach the performance of radix-4. While there are still some optimizations possible for radix-2, we are fairly sure that radix-4 will keep performing better.

Future work. NTT is a complex algorithm. Combining this with the fact that it is very similar to DFT means that there

are still countless ideas that can be applied and explored. There are a few promising concepts that we could not consider due to time (or hardware) constraints: using AVX-512 intrinsics, which would solve the bottleneck we are facing when computing the vector-modulo operation (while also doubling the throughput of all other vector operations); exploring other algorithms e.g. radix-8, radix-16, and/or different combinations of these; experimenting with code generation and auto tuning; and finally an in-depth analysis and evaluation of different input reordering strategies to improve cache locality.

6. CONTRIBUTIONS OF TEAM MEMBERS

Rudolf. Implemented the basic scalar implementations of all algorithms, increased precomputation (twiddle factors & more), vectorized radix-4 (worked with Artur), did the profiling and roofline analysis to direct optimizations at bottlenecks.

Artur. Implemented the basic scalar implementations of all algorithms, increased precomputation (twiddle factors & more), vectorized radix-4 (worked with Rudolf), tried cache optimization by reordering, improved all implementations with precomputed floored division.

Rijad. Vectorized radix-2 scalar, tested several implementations of the modulo operations, split up the best implementation (scalar and vector) of radix-4 to distinguish the speed-up of each improvement.

Philippe. Optimized radix-2 scalar, discovered how to vectorize the modulo operation, split up the best implementations (scalar and vector) of radix-2 to distinguish the speed-up of each improvement, did the cost analysis, and made performance plots.

7. REFERENCES

[1] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[2] Patrick Longa and Michael Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," *TCANS*, 2016.

[3] S. Boussakta and A. G. J. Holt, "Number theoretic transforms and their applications in image processing," *Advances in Imaging and Electron Physics*, 1999.

[4] Franz Franchetti and Markus Püschel, "Fast fourier transform," *Encyclopedia of Parallel Computing*, 2011.

[5] Matteo Frigo and Steven G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, pp. 216–231, 2005.

[6] Spiral web site, " `http://spiral.net`.

[7] David Harvey, "Faster arithmetic for number-theoretic transforms," *Journal of Symbolic Computation*, 2014.

[8] Gregor Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography," *IACR Cryptology*, 2018.

[9] C. Aguilar-Melchor, J. Barrier, S. Guelton, Ad. Guinet, M. Killijian, and T. Lepoint, "Nfllib: Ntt-based fast lattice library," *RSA Conference Cryptographers' Track*, 2016.

[10] A. Mohsen, M. Sobh, and A. Bahaa-Eldin, "Performance analysis of number theoretic transform for lattice-based cryptography," *ICCES*, 2018.

[11] Apfloat web site about NTT, " `http://www.apfloat.org/ntt.html`.

[12] James W. Cooley and John W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[13] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery, *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press, 2007.

[14] S. Chellappa, F. Franchetti, and M. Püschel, "How to write fast numerical code: A small introduction," *GTTSE*, vol. 5235, pp. 196–259, 2008.

[15] David H Bailey, "The computation of π to 29,360,000 decimal digits using borweins' quartically convergent algorithm," *Mathematics of Computation*, vol. 50, no. 181, pp. 283–296, 1988.

[16] Joachim von zur Gathen and Jrgen Gerhard, *Modern Computer Algebra Chapter 2.2*, Cambridge University Press, USA, 3rd edition, 2013.

[17] Donald Ervin Knuth, *The art of computer programming*, vol. 3, Pearson Education, 1997.

[18] SymPy Development Team, " `https://docs.sympy.org/latest/modules/discrete.html#number-theoretic-transform`.

[19] Intel, " `https://software.intel.com/content/www/us/en/develop/tools.html`.

[20] Shekhar Suresh Chandra, " `http://finitetransform.sourceforge.net/`.

[21] V. Elango, N. Sedaghati, F. Rastello, L.-N. Pouchet, J. Ramanujam, R. Teodorescu, and P. Sadayappan, "Augmenting the roofline model via lower bounds on data movement," Tech. Rep., OSU-CISRC, 2014.

[22] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel, "Applying the roofline model," *ISPASS*, 2014.

[23] Terpstra D., Jagode H., You H., and Dongarra J., "Collecting performance data with papi-c," *Tools for High Performance Computing 2009*, 2010.