Vectorising and distributing NTTs to count Goldbach partitions on Arm-based supercomputers

Ricardo Jesus
EPCC
The University of Edinburgh
Edinburgh, United Kingdom
rjj@ed.ac.uk

Tomás Oliveira e Silva
IEETA/DETI
Universidade de Aveiro
Aveiro, Portugal
tos@ua.pt

Michèle Weiland
EPCC
The University of Edinburgh
Edinburgh, United Kingdom
m.weiland@epcc.ed.ac.uk

***Abstract*—In this paper we explore the usage of SVE to vectorise number-theoretic transforms (NTTs). In particular, we show that 64-bit modular arithmetic operations, including modular multiplication, can be efficiently implemented with SVE instructions. The vectorisation of NTT loops and kernels involving 64-bit modular operations was not possible in previous Arm-based SIMD architectures, since these architectures lacked crucial instructions to efficiently implement modular multiplication. We test and evaluate our SVE implementation on the A64FX processor in an HPE Apollo 80 system. Furthermore, we implement a distributed NTT for the computation of large-scale exact integer convolutions. We evaluate this transform on HPE Apollo 70, Cray XC50, and HPE Apollo 80 systems, where we demonstrate good scalability to thousands of cores. Finally, we describe how these methods can be utilised to count the number of Goldbach partitions of all even numbers to large limits. We present some preliminary results concerning this problem, in particular a histogram of the number of Goldbach partitions of the even numbers up to 2^{40}.**

***Index Terms*—A64FX; Arm; Goldbach partitions; Modular multiplication; NTT; SVE; ThunderX2; Vectorisation.**

I. Introduction

Despite having made their debut on the HPC scene only a few years ago, Arm-based processors have already become some of the most powerful chips in the world. Advanced features of recent Arm architectures, such as the Scalable Vector Extension (SVE), have enabled new optimisations that until very recently were not possible. One of these optimisations is the vectorisation of loops containing modular arithmetic operations, in particular 64-bit modular multiplication. Modular arithmetic can be viewed as conventional integer arithmetic, where basic operations such as additions, subtractions and multiplications are taken modulo an integer (the modulus). In general, the most challenging of the elementary modular operations is modular multiplication, since it involves a non-trivial modular reduction step. There are a few well-known methods for performing fast modular multiplications, such as the Montgomery multiplication method [1] that we employ in this work. However, previous single instruction multiple data (SIMD) architectures, namely Neon, lacked crucial instructions to implement these methods efficiently (for example, instructions that calculate the "full" 128-bit result of a 64-bit multiplication). Because of this, modular multiplications have frequently hindered the vectorisation of loops they are a part of.

One place where loops with several modular operations occur is in number-theoretic transforms (NTTs). NTTs are analogous to the discrete Fourier transform (DFT), with the key difference that all arithmetic operations proceed over a ring or field instead of the complex domain [2, §9.5.5]. In practice, this typically means replacing the conventional additions, subtractions and multiplications in a DFT by their modular counterparts, where the modulus is usually a prime number satisfying some conditions. Importantly, due to the underlying modular arithmetic, NTT computations are exact. As a result, NTTs have become an important tool to compute the *exact* convolution of arbitrarily large integer sequences, avoiding the rounding and representation errors that arise in conventional DFTs. Incidentally, NTTs are usually more computationally demanding than DFTs, since they require more operations to implement the modular arithmetic, and some of those operations *per se* (such as the modular reductions in multiplications) make other optimisations such as vectorisation more difficult, as we have previously mentioned.

In this paper we explore the usage of SVE to optimise 64-bit modular arithmetic operations in NTTs, with a special focus on modular multiplications. We show that it is now possible to vectorise loops with modular multiplications efficiently by utilising SVE, and choosing the parameters for the Montgomery method in such a way that only shifts and arithmetic modulo 2^{64} are used in specific junctures of the algorithm. It is worth noting that none of the compilers tested by us (i.e. Arm, Cray, Fujitsu and Gnu) were able to vectorise these loops automatically. We evaluate our implementation on the A64FX processor.

Furthermore, we implement a distributed large-scale NTT for the exact computation of large integer convolutions. Our implementation is based on Bailey's four-step algorithm [3] and uses a hybrid MPI and OpenMP parallelisation approach. We evaluate our implementation on three Arm-based HPC systems:

Fulhame An HPE Apollo 70 cluster that consists of 64 dual-socket compute nodes connected with Mellanox EDR Infiniband (IB). Each node contains two 32-core Marvell

ThunderX2 processors running at 2.2GHz and 256GB of DDR4 random-access memory.

Isambard 1 A Cray XC50 system, also based on Marvell ThunderX2 processors, but containing 329 compute nodes connected with Cray Aries (169 of which have 512GB of RAM, the rest have 256GB).

Isambard 2 An HPE Apollo 80 system based on Fujitsu A64FX processors. It consists of 72 nodes connected with Mellanox Infiniband, each node containing a 48-core A64FX processor running at 1.8 GHz.

Our results (Figs. 3 and 4) show excellent strong and weak scaling to a few thousand cores, with the percentage of non-overlapped MPI communication being kept below 4% when using 32 nodes of Fulhame (2048 cores in total) to compute large transforms of 2^{39} points.

Finally, we employ the aforementioned methods to compute some preliminary results concerning the number of Goldbach partitions of all even integers to a large limit. Goldbach partitions correspond to the different ways in which an even integer n can be represented as the sum of two prime numbers (the Goldbach conjecture is equivalent to the statement that all even integers greater than 2 have at least one such representation). We plot the number of Goldbach partitions of the even numbers up to 2^{40} (Fig. 5), which corresponds to an improvement of more than 2000x over the previously published record [4]. Our aim is to utilise these methods to carry out a much larger-scale computation of Goldbach partitions in the near future.

Although we specifically discuss our work in the context of counting Goldbach partitions, our contributions are more widely applicable to modular arithmetic operations, which are fundamental to pure mathematics and number theory applications. In particular our contributions are:

1) The demonstration of the use of SVE to successfully vectorise loops with 64-bit modular multiplications;
2) The implementation of local and distributed number-theoretic transform methods, with applications to a wide spectrum of problems that require exact convolution of (very large) integer sequences;
3) The performance evaluation of these methods on Arm-based supercomputers.

The remainder of this paper is organised as follows. In Sec. II we present the SVE implementation of the modular arithmetic operations we employ in NTTs. In Sec. III we discuss the implementation of NTTs themselves, namely the algorithm we use to compute them in a distributed fashion. In Sec. IV we describe how these methods are utilised to compute Goldach partitions to large limits. We evaluate the methods we have developed in Sec. V, and we discuss related work in Sec. VI. Section VII concludes the paper.

II. Modular arithmetic with SVE

In this section we present SVE implementations of the three basic modular arithmetic operations required for number-theoretic transforms (NTTs): addition, subtraction, and multiplication. Other operations, such as modular exponentiation, are implemented resorting to these elementary operations, and thus they are not considered here. As we shall see below, the only operation that poses an actual challenge for efficient implementation, particularly with vector instructions, is modular multiplication.

Although our motivation for developing vector versions of these modular operations is to vectorise the butterfly loops of NTTs, the same techniques can also be employed in other codes—for example, codes that utilise the Chinese Remainder Theorem (CRT) to work with a very large modulus broken into several smaller moduli, with all operations taken modulo each of those moduli.

Henceforth `umod_t` is used as an alias for an unsigned 64-bit integer type, and N is the modulus over which the operations are taken (a constant of `umod_t` type). We assume throughout that $N \geq 2^{32}$, since different strategies could be used to implement these operations more efficiently for smaller integer types. For each operation we present a generic implementation written in C, followed by an SVE version implemented using SVE intrinsics [5].

A. Addition and subtraction

Addition and subtraction of two integers a and b modulo a third integer N may be expressed as

$$\texttt{add_mod}\,(a, b) = (a + b) \bmod N, \tag{1}$$

$$\texttt{sub_mod}\,(a, b) = (a - b) \bmod N. \tag{2}$$

When the operands a and b are given in their reduced form modulo N, i.e. $0 \leq a, b < N$, the implementation of these operations is straightforward:

```
1 inline umod_t add_mod(umod_t a, umod_t b)
2 {
3     umod_t z = a+b;
4     return z >= N ? z-N : z;
5 }
```

```
1 inline umod_t sub_mod(umod_t a, umod_t b)
2 {
3     umod_t z = a-b;
4     return a >= b ? z : z+N;
5 }
```

In the code above it is assumed that $N \leq 2^{63}$, otherwise the computation of z in the function `add_mod` may overflow. If $N > 2^{63}$, one can instead compute modular addition via a modular subtraction between a and $N - b$.

Since these implementations only utilise additions, subtractions, and conditionals, they can be readily rewritten utilising SVE intrinsics:

```
1 inline svuint64_t add_mod_sve(
2     svbool_t pg, svuint64_t a, svuint64_t b)
3 {
4     svuint64_t z = svadd_x(pg, a, b);   // z = a+b
5     svbool_t pc = svcmpge(pg, z, N); // c = z >= N
6     return svsub_m(pc, z, N);      // if(c) z -= N
7 }
```

```
1 inline svuint64_t sub_mod_sve(
2     svbool_t pg, svuint64_t a, svuint64_t b)
3 {
```

```
4     svuint64_t z = svsub_x(pg, a, b);   // z = a-b
5     svbool_t pc = svcmplt(pg, a, b);  // c = a < b
6     return svadd_m(pc, z, N);       // if(c) z += N
7 }
```

The compilers we tested (Arm, Cray, Fujitsu, and GNU) managed to vectorise loops with modular additions and subtractions automatically, *when these operations were used in isolation*. However, in practice almost all loops involving modular arithmetic include some form of modular multiplication, and once modular multiplication is present none of these compilers were able to vectorise such loops automatically. For this reason, the SVE versions of modular addition and subtraction presented above are necessary to glue together the SVE-based modular multiplication we present next with the rest of the code.

B. Multiplication

Modular multiplication is an operation far more expensive than modular addition or subtraction. Formally, we want to compute

$$\texttt{mul_mod}\,(a, b) = (a \cdot b) \bmod N, \tag{3}$$

where N, as before, is a 64-bit integer. The extra cost of modular multiplications arises due to the more costly modular reduction step that needs to be performed, which in the case of modular addition and subtraction can be effected trivially via a conditional subtraction or addition.

A possible naïve modular multiplication implementation might look like this (once again assuming $0 \leq a, b < N$):

```
1 inline umod_t mul_mod_naive(umod_t a, umod_t b)
2 {
3     return (a*b) % N;
4 }
```

However, this approach is flawed in two fundamental ways. Firstly, N is effectively restricted to half the size of the `umod_t` type, or else the intermediate product of a and b may overflow. Since `umod_t` is a 64-bit unsigned integer type, this implies $N \leq 2^{32}$, which is not practical for our purposes where we need N to be large. Secondly, computing the remainder of the product of `a` and `b` by N directly is, in virtually all CPUs, very expensive for all but a small selection of values of N. For example, in the A64FX processor a remainder operation with 64-bit unsigned operands is typically implemented via an unsigned division followed by a fused multiply-subtract. The division operation alone takes between 9–41 cycles, and it cannot be pipelined [6]. On top of that, several of these divisions need to be executed to compute the remainder of a 128-bit dividend and 64-bit divisor—the actual case we are interested in as we shall see next. For these reasons, no nontrivial division or modulo operations should take part in an efficient implementation of modular multiplication.

The first issue can be circumvented by computing the full 128-bit product of the two 64-bit operands, and only then taking the remainder of the 128-bit result by N. Computing 128-bit products can be achieved either by using 128-bit integer arithmetic (where supported), or with a mix of integer and floating-point arithmetic (in the latter case, a floating-point multiplication is used to compute the high part of the product of two numbers, and thus N becomes limited by the size of the mantissa of the floating-point type used). An approach using only integer arithmetic is usually simpler and more efficient, as it avoids conversions from integer to floating-point representations and back, and for this reason it is the strategy we use. For an example of a mixed integer and floating-point approach refer to [7, §39.1]. For compilers that support 128-bit integer types, the following C function-like macro can be used to compute the 128-bit product of two 64-bit integers:

```
1 // u128 is a 128-bit unsigned integer type
2 #define umul128(ph, pl, m0, m1) do {   \
3     u128 __p = (u128)(m0)*(m1);       \
4     (ph) = __p >> 64;                 \
5     (pl) = __p;                       \
6 } while(0)
```

Alternatively, for AArch64 targets the macro may be implemented utilising inline assembly has follows:

```
1 #define umul128(ph, pl, m0, m1) do {      \
2     uint64_t __m0 = (m0), __m1 = (m1);    \
3     __asm__ ("umulh\t%0,␣%1,␣%2"          \
4              : "=r" (ph)                  \
5              : "r" (__m0), "r" (__m1));   \
6     (pl) = __m0 * __m1;                   \
7 } while(0)
```

Meanwhile, we can avoid taking the remainder by N explicitly by utilising an efficient modular reduction algorithm. The most common strategies for this are the long divide algorithm of Knuth [8, Algorithm 4.3.1D], and the Barrett [9] and Montgomery methods [1]. All these strategies involve some pre-calculation dependent on the modulus N, and consequently they are most effective when several modular multiplications have to be performed for a fixed modulus (as is the case with NTTs or, for example, power ladders in modular exponentiation). Here we have chosen to use the Montgomery method, as it has been been reported to be slightly faster than the other strategies [10], [11], and as we shall see below, it lends itself well to an SVE-based implementation.

We now briefly describe the Montgomery method. For a more extensive discussion and proofs refer to the original paper [1].

Let R be an integer such that $R > N$ and $\gcd(N, R) = 1$, and define $N' = \left(-N^{-1}\right) \bmod R$ (N' being one of the pre-computed constants required by this method). The idea is to, instead of working with numbers modulo N directly, work in a different modular representation where a particular reduction step exists and that is "easy" to compute (compared to the naïve modulo N reduction). In this sense, let $\tilde{x}$ denote the Montgomery representation of x, defined as

$$\tilde{x} = (xR) \bmod N. \tag{4}$$

Then, for any integer T we have the reduction

$$\mathrm{REDC}(T) = \left[T + N\left((TN') \bmod R\right)\right]/R \bmod N, \tag{5}$$

with

$$\mathrm{REDC}(T) \equiv TR^{-1} \pmod{N}. \tag{6}$$

Moreover, when $0 \leq T < RN$, either the reduction (5) with the mod N operation omitted equals $\left(TR^{-1}\right) \bmod N$, or it exceeds it by N. Crucially, if we let $T = \tilde{x}\tilde{y}$ we obtain $\mathrm{REDC}(T) = \mathrm{REDC}(\tilde{x}\tilde{y}) \equiv (\tilde{x}\tilde{y})\,R^{-1} \equiv (xR)\,(yR)\,R^{-1} \equiv xyR \equiv \widetilde{xy} \pmod N$, and since $0 \leq T < N^2$, the mod N operation in (5) can be replaced with a conditional subtraction. In other words, the reduction (5), *without the mod N operation*, plus a trivial adjustment step can be used to reduce the product of two numbers in Montgomery form to the Montgomery form of their product modulo N, with the only division or modulo operations required having R as divisor. Provided division and modulo by R can be computed efficiently (see below), this strategy tends to be significantly faster than calculating the remainder by N directly. As we have mentioned previously, this is especially true when several modular products are to be performed over the same modulus, since in this case the overhead of calculating the auxiliary constant N' and of moving back and forth between standard and Montgomery representations can be amortised over the several multiplications that have to be performed. Finally, it should be noted that the addition and subtraction modulo N of two operands in Montgomery form equals the Montgomery form of their sum and difference, i.e.

$$\widetilde{x \pm y} = (\tilde{x} \pm \tilde{y}) \bmod N, \tag{7}$$

and so the routines we have presented previously for effecting these operations require no modification whatsoever to work with numbers in Montgomery form.

Assuming $N \geq 3$ and odd, as is the case in nearly all cases of practical interest (usually N is a large prime as we shall see later), R can be chosen to be any sufficiently large power of two—thus rendering the division and mod operations in (5) trivial. It turns out that the choice $R = 2^{64}$ is especially suitable for an SVE implementation, since it simplifies the intermediate calculations that have to be performed considerably. With this choice of R and assuming N' has been precomputed, the modular product of two integers (in Montgomery form), via reduction (5), can be implemented in C as follows:

```
1 inline umod_t mul_mod(umod_t a, umod_t b)
2 {
3     umod_t m, xh, xl, yh, yl, z;
4     umul128(xh, xl, a, b);      // x = a*b
5     m = (umod_t)(xl*N_prime);   // m = (x*N') mod R
6     umul128(yh, yl, m, N);      // y = m*N
7     z = xh+yh+((xl+yl) < xl);   // z = (x+y)/R
8     return z >= N ? z-N : z;
9 }
```

This implementation has been purposely written in a way that can be easily translated into SVE. The key observation is that, with $R = 2^{64}$, the operations on the low and high parts of the 128-bit products that are calculated (x and y in the code) are carried out almost independently, with the exception of the carry for determining `z`—but this carry can easily be accounted for with an extra overflow check. Putting everything together, we obtain the following SVE code:

```
1 inline svuint64_t mul_mod_sve(
2     svbool_t pg, svuint64_t a, svuint64_t b)
3 {
4     svbool_t pc;
5     svuint64_t m, xh, xl, z;
6
7     /* x = a*b */
8     xl = svmul_x(pg, a, b);
9     xh = svmulh_x(pg, a, b);
10
11    /* m = (x*N') mod R */
12    m = svmul_x(pg, xl, N_prime);
13
14    /* c = (x mod R)+(m*N mod R) >= R */
15    pc = svcmpgt(pg, xl, svmla_x(pg, xl, m, N));
16
17    /* z = (x+m*N)/R + c */
18    z = svadd_x(pg, xh, svmulh_x(pg, m, N));
19    z = svadd_m(pc, z, (uint64_t)1);
20
21    /* adjust result */
22    return svsub_m(svcmpge(pg, z, N), z, N);
23 }
```

Finally, we need to consider how to efficiently move back and forth between standard and Montgomery representations. Formally, we want to compute

$$\text{to_mont}(x) = (xR) \bmod N = \tilde{x} \tag{8}$$

$$\text{from_mont}\,(\tilde{x}) = \left(\tilde{x}R^{-1}\right) \bmod N = x. \tag{9}$$

Using an extra auxiliary constant $Q = R^2 \bmod N$ and resorting to the `mul_mod` routine presented above (or its SVE version), these operations can be implemented efficiently via

$$\text{to_mont}(x) = \text{mul_mod}(x, Q), \tag{10}$$

$$\text{from_mont}(\tilde{x}) = \text{mul_mod}(\tilde{x}, 1). \tag{11}$$

Evidently, in the case of `from_mont` further optimisations are possible by expanding the `mul_mod` call and removing the operations that become redundant due to the superfluous multiplication by one.

We note that, for the computations described in Sec. IV, we do not need to transform data to and from Montgomery forms explicitly—i.e. we never actually need to call the conversion routines above. On the one hand, we can trivially generate the inputs for our computations directly in Montgomery form (see Sec. IV-A). On the other hand, since the last modular multiplication we perform when computing intermediate results is a multiplication by a constant (the final normalisation in the inverse transform of Sec. III-D), we can combine this multiplication with the conversion from Montgomery to standard representations by *not* converting the constant to Montgomery representation beforehand. This happens because if we reduce the product of a number in Montgomery form ($\tilde{x}$) with a number in standard form (y) we get

$$\mathrm{REDC}(\tilde{x}y) = ((xR)y)R^{-1} \bmod N \tag{12}$$

$$= xy \bmod N, \tag{13}$$

which is precisely the modular product of x and y *in standard representation*.

III. Distributed number-theoretic transforms

In this section we present the implementation of a distributed number-theoretic transform (NTT) with applications

to the computation of very large integer convolutions. Furthermore, we describe how the SVE routines for modular arithmetic presented in Sec. II are utilised to optimise the local transforms performed throughout the distributed computation.

A. Background

Number-theoretic transforms are a generalisation of the discrete Fourier transform (DFT) obtained by performing the transform over a ring or field instead of the usual complex domain, and recasting the arithmetic operations of the transform in terms of the new domain [2, §9.5.5]. In practice, this usually means replacing the conventional additions, subtractions, and multiplications in a DFT by their modular counterparts, whereby the modulus, usually a prime number, has to satisfy some properties. First, for a NTT of length m in the ring of integers modulo n, there needs to be a m-th root of unity modulo n (a number r is called a primitive m-th root of unity modulo n if $r^k \not\equiv 1 \pmod{n}$ for $1 \leq k < m$ and $r^m \equiv 1 \pmod{n}$). Secondly, for the inverse transform, m has to invertible modulo n, which implies that m and n must be coprime.

In our work we restrict the choice of modulus to the prime numbers. In particular, for a NTT of length m we choose a prime modulus p of the form $p = km + 1$ for some positive integer k, whereby an appropriate primitive m-th root of unity g can be found as per [7, §39.6]. Then, the (forward) NTT of a sequence of length m can be expressed as

$$X_k = \sum_{j=0}^{m-1} x_j g^{-jk} \bmod p, \tag{14}$$

whilst the respect inverse transform is given by

$$x_j = m^{-1} \sum_{k=0}^{m-1} X_k g^{jk} \bmod p, \tag{15}$$

where m^{-1} is the multiplicative inverse of m mod p (which under our choice of modulus is guaranteed to exist).

Given the similarities between NTTs and the DFT, NTTs can in general be computed with efficient FFT-based algorithms by simply translating those algorithms into the domain of the NTT. Hence, just like the DFT, NTTs can be computed in $O(m \log m)$ time (m being the length of the transform). Unlike the DFT, however, NTTs do not transform to a meaningful "frequency" domain. Despite this, they do can be utilised to effect integer convolutions via the convolution theorem [2, Theorem 9.5.11]:

$$x \times y = \mathrm{NTT}^{-1}\left(\mathrm{NTT}(x) * \mathrm{NTT}(y)\right), \tag{16}$$

where $x \times y$ is the cyclic convolution of the integer sequences x and y (of the same length), and $*$ denotes element-wise multiplication. Since with the underlying modular arithmetic no rounding errors occur, the main application of NTTs is the computation of fast *exact* integer convolutions. Incidentally, the computation of NTTs is inherently more computationally demanding than that of conventional DFTs, mainly because the former require more arithmetic operations to effect the modular arithmetic, and because those modular operations (especially the modular reductions) complicate common optimisations such as vectorisation. For these reasons, despite their practical utility, NTTs are often avoided unless when exact results are paramount.

B. Outline of computation

In this section we present a simplified top-down view of the algorithm we use to compute distributed NTTs. The goal is to outline the steps we will carry out, whilst deferring the details of the computation to subsequent sections. The skeleton of the algorithm follows Bailey's "four-step" algorithm for hierarchical memory systems [3]. In this sense, let $m = H \times W$ be the size of the array x whose NTT we wish to compute, m a power of two. In practice, x will be distributed amongst several processes, but we will ignore this aspect for the time being. The algorithm proceeds as follows:

1) Let A be a $H \times W$ "matrix view" of x, where $A_{j,k}$ corresponds to x_{jW+k}, with $0 \leq j < H$ and $0 \leq k < W$ (i.e. x is organised in row-major order in A);
2) Replace each column $A_{:,k}$ with its transform $\mathrm{NTT}(A_{:,k})$
3) Multiply each element $A_{j,k}$ by g^{-jk}, g a primitive m-th root of unity in the appropriate domain;
4) Replace each row $A_{j,:}$ with its transform $\mathrm{NTT}(A_{j,:})$

After these steps, the transform elements $\{X_k\}_{k=0}^{m-1}$ are the elements of A read in column-major order (i.e., after the transform, $A_{j,k}$ corresponds to X_{j+kH}).

This process computes the *forward* NTT of x. The inverse transform can be obtained by replacing the forward NTT calls in steps 2 and 4 with inverse NTTs, using g^{jk} in step 3 instead of g^{-jk}, and following steps 2 to 4 in reverse order, assuming the transformed elements X_k are organised in column-major order in A as per the output of the forward transform. The output of the inverse transform will be, once again, in row-major order.

C. Local transforms

In this section we present the "FFT-based" algorithms we use to compute local NTTs. These algorithms are utilised in steps 2 and 4 of Sec. III-B to compute NTTs on the rows and columns of the matrix mentioned therein.

1) Stockham algorithm: The Stockham algorithm is a method for computing the FFT of a signal *in-order* and with an innermost loop with *unit stride memory accesses* [12], [13]. Because the transform is computed in-order, the Stockham algorithm does not require the bit-scrambling procedures used by the Cooley-Tukey and Gentleman-Sande methods (see next section). The cost of these properties is that one must use an extra copy of the data.

The Stockham algorithm can be readily adapted to NTTs. In fact, since its innermost loop runs sequentially through memory, it is fairly straightforward to vectorise it. In pseudocode, we have:

Algorithm 1 (Stockham NTT.) Given an array a of l integers and an auxiliary buffer b of the same size, compute the NTT

of a. We use C syntax for pointers a and b, so that $\&a[k]$ is the address of the element of a at index k. All modular operations are performed over an implicit modulus N, and g is a suitable primitive root of unity.

$a' \leftarrow a$; ▷ Pointer assigment
for($k \leftarrow 1, half \leftarrow l/2$; $k \leq half$; $k \leftarrow 2k$) {
  for($m \leftarrow 0$; $m < half$; $m \leftarrow m + k$)
    stockham_butterflies(
      $\&a[m]$, $\&a[m + half]$,
      $\&b[2m]$, $\&b[2m + k]$,
      g^{-m}, k); ▷ g^m for denormalised NTT^{-1}
  $a, b \leftarrow b, a$; ▷ Swap pointers
}
if($a \neq a'$) for($0 \leq i < l$) $a'[i] \leftarrow a[i]$;

The "stockham_butterflies" routine can be implemented utilising the SVE methods described in Sec. II as follows (we assume all operands are given in Montgomery form):

```
inline void stockham_butterflies(
    umod_t const * restrict al,
    umod_t const * restrict ah,
    umod_t       * restrict bl,
    umod_t       * restrict bh,
    umod_t w, ulong k)
{
    ulong j = 0;
    svbool_t pg = svwhilelt_b64(j, k);
    svuint64_t al_vec, ah_vec, bl_vec, bh_vec;
    svuint64_t w_vec = svdup_u64(w);

    do {
        al_vec = svld1(pg, al+j);
        ah_vec = svld1(pg, ah+j);

        bl_vec = add_mod_sve(pg, al_vec, ah_vec);
        bh_vec = sub_mod_sve(pg, al_vec, ah_vec);
        bh_vec = mul_mod_sve(pg,  w_vec, bh_vec);

        svst1(pg, bl+j, bl_vec);
        svst1(pg, bh+j, bh_vec);

        j += svcntd(), pg = svwhilelt_b64(j, k);
    } while(j < k);
}
```

2) Cooley-Tukey and Gentleman-Sande NTTs: The Cooley-Tukey and Gentleman-Sande algorithms are two analogous methods for computing the FFT of a signal [14]. Unlike the Stockham algorithm they work *in-place*, but require *bit-scrambling* routines to reshuffle the data at the beginning (Cooley-Tukey) or at the end (Gentleman-Sande) of the computation. Moreover, they access data with *power-of-two strides* in their innermost loops. The need to perform bit-scrambling and the power-of-two memory accesses tend to make these routines less efficient than the Stockham algorithm. However, due to how we employ them to compute convolutions in Alg. 4, in practice we can circumvent both of these issues.

First, if one performs a transform using the Gentleman-Sande algorithm, followed by one using the Cooley-Tukey algorithm, the scrambling routines become superfluous because they cancel each other out. Because of this, when computing a convolution (which requires the computation of forward followed by inverse transforms, see (16)), we use Gentleman-Sande form for the forward NTTs, and Cooley-Tukey form for the inverse NTT, in both cases omitting the scrambling procedures.

Secondly, in practice we use these algorithms to compute not just one but *several* transforms arranged in a certain way, *all at once*. This allows us to avoid the aforementioned memory locality issues. The data layout we use for this computation is the one discussed in stage ③ of Sec. III-D, whereby we have a matrix $r \times l$, stored by column, and we wish to compute a transform on each row of the matrix. By computing the r transforms all at once, we effectively perform all operations with unit stride memory accesses. Furthermore, we can efficiently parallelise this computation by having different threads compute butterflies concurrently in each stage of the transform. Putting it all together, we have the following pseudocode:

Algorithm 2 (Cooley-Tukey and Gentleman-Sande NTTs.) Given a matrix a with dimensions $r \times l$ stored by column, compute the NTT of each row using the Cooley-Tukey or Gentleman-Sande algorithm. The output (input) of the Gentleman-Sande (Cooley-Tukey) method is in bit-scrambled order. OpenMP is used to illustrate how the computation can be parallelised. As in Alg. 1, we use C-based pointer syntax for a. If g^{jv} is used instead of g^{-jv}, these algorithms compute denormalised inverse NTTs.

1: (Cooley-Tukey.)
for($k \leftarrow 1, v \leftarrow l/2$; $k < l$; $k \leftarrow 2k, v \leftarrow v/2$) {
  #pragma omp for schedule(dynamic)
  for($h \leftarrow 0$; $h < l/2$; $h \leftarrow h + 1$) {
    $i \leftarrow \lfloor h/k \rfloor, j \leftarrow h \bmod k$;
    $s \leftarrow jr + 2ikr, t \leftarrow s + kr$;
    twist_butterfly($\&a[s]$, $\&a[t]$, g^{-jv}, r);
  }
}

2: (Gentleman-Sande.)
for($k \leftarrow l/2, v \leftarrow 1$; $k > 0$; $k \leftarrow k/2, v \leftarrow 2v$) {
  #pragma omp for schedule(dynamic)
  for($h \leftarrow 0$; $h < l/2$; $h \leftarrow h + 1$) {
    $i \leftarrow \lfloor h/k \rfloor, j \leftarrow h \bmod k$;
    $s \leftarrow jr + 2ikr, t \leftarrow s + kr$;
    butterfly_twist($\&a[s]$, $\&a[t]$, g^{-jv}, r);
  }
}

The routine "twist_butterfly" can be implemented using the SVE-based modular arithmetic routines of Sec. II as follows:

```
inline void twist_butterfly(
  umod_t * restrict a,
  umod_t * restrict b,
  umod_t w, ulong r)
{
  ulong j = 0;
  svbool_t pg = svwhilelt_b64(j, r);
  svuint64_t a_vec, b_vec, w_vec = svdup_u64(w);

  do {
```

```
11      b_vec = mul_mod_sve(pg, svld1(pg, b+j), w_vec);
12      a_vec = svld1(pg, a+j);
13
14      svst1(pg, a+j, add_mod_sve(pg, a_vec, b_vec));
15      svst1(pg, b+j, sub_mod_sve(pg, a_vec, b_vec));
16
17      j += svcntd(), pg = svwhilelt_b64(j, r);
18    } while(j < r);
19  }
```

Analogously, “butterfly_twist” can be implemented as “twist_butterfly”, but replacing lines 11–15 with:

```
1       a_vec = svld1(pg, a+j);
2       b_vec = svld1(pg, b+j);
3
4       svst1(pg, a+j, add_mod_sve(pg, a_vec, b_vec));
5       b_vec = sub_mod_sve(pg, a_vec, b_vec);
6       svst1(pg, b+j, mul_mod_sve(pg, b_vec, w_vec));
```

D. Distribution and parallelisation

As we have mentioned previously, our main goal with the utilisation of NTTs is to compute the convolution of very large integer sequences. In practice, these sequences are so large that we need the aggregated memory of several compute nodes to store them. Hence, the steps described in Sec. III-B to compute a NTT have to be carried out in a distributed fashion. In this section we describe how this computation is performed.

Our implementation uses a mix of MPI and OpenMP, the former for interprocess communication, and the latter for (intraprocess) parallelisation. In general, we are aiming for a scenario where we run one MPI process per NUMA domain, and as many OpenMP threads per MPI process as possible so that, in total, we have one OpenMP thread per core (and an equal amount of OpenMP threads per MPI process). In practice, we also bind all OpenMP threads to cores and place them such that they are as close as possible in hardware for each MPI process (e.g. belonging to the same NUMA domain). Only the master thread of an MPI process issues MPI calls.

As before, let x be an integer array of length $m = H \times W$ whose NTT we wish to compute, m being a power of two. Let A be a matrix-view of x with dimensions $H \times W$, where element $A_{j,k}$ corresponds to x_{jW+k}, with $0 \leq j < H$ and $0 \leq k < W$ (i.e. A is a view of x organised in row-major order). Furthermore, let P be the number of processes over which the computation is distributed. For simplicity, we assume $\min(H, W)$ is divisible by P, so that all processes hold portions of x of equal sizes. Finally, let $h = H/P$ and $w = W/P$.

Initially, each process starts with a group of w adjacent columns of A, i.e. one process contains columns numbered $[0, w)$, another $[w, 2w)$, and so on. The data is stored column by column, so that two elements $A_{j,k}$ and $A_{j+1,k}$ reside at adjacent memory addresses. This corresponds to the region marked by ① in Fig. 1. Then, the computation proceeds in the three major stages described below (all operations are implicitly carried out modulo a suitable number N, with g a m-th root of unity modulo N).

① (Column stage.) In this stage, the goal of each process is to compute the NTT of each of its columns (step 2 of Sec. III-B), apply twiddle factors (step 3 of Sec. III-B), and exchange data with other process in preparation for the row transforms that are going to be performed in the final stage of the computation. Let $A_{:,k}$ denote the k-th column of A and p, $0 \leq p < P$, the index/rank of a process. Then, in pseudocode, each process executes:

for($pw \leq k < pw + w$) {

1: (Compute NTT of column.)
   $A_{:,k} \leftarrow \mathrm{NTT}(A_{:,k})$; ▷ via Alg. 1

2: (Apply twiddle factors.)
   for($0 \leq j < H$) $A_{jk} \leftarrow A_{jk} g^{-jk}$;

3: (Exchange data.) Each process sends elements $[0, h)$ of their column to process 0, elements $[h, 2h)$ to process 1, and so on. The exchange is made in-place, and only the master thread issues this exchange (see below).
   alltoall($A_{:,k}$, h);

}

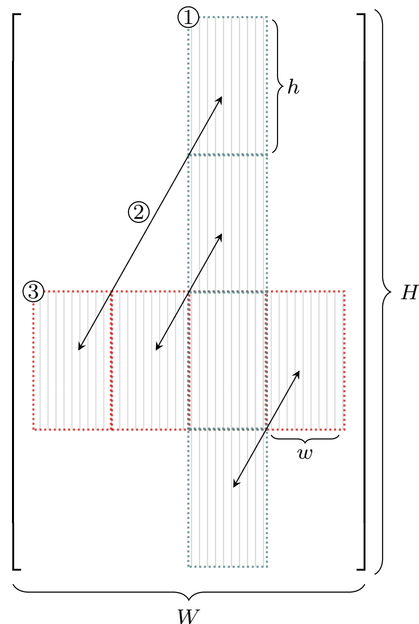

Figure 1. Computation of a distributed NTT, depicted for $P = 4$ processors (description in text).

In the code we process each column one by one for illustration purposes. In practice, however, the computation is carried in tranches of C columns at a time (C a user defined parameter). For each tranche, the C columns are processed in parallel by the threads of a process: each thread computes the NTT of a column, applies the respective twiddle factors, and moves on to the next column that has not yet processed. Once the C columns have been processed, the master thread of the process issues an all-to-all call to exchange C segments of length h, one segment per column, with each of the other processes. Process 0 gets the first h elements of the

columns, process 1 the next h, and so on, with the data being exchanged in-place (i.e. the data sent to a process is replaced with the data received from it). Whilst this communication is happening, the remaining threads start processing the next tranche, and so forth. This allows us to overlap the MPI commutation with computations.

At the end of this stage, all columns will have been processed as described above and the large blocks depicted in Fig. 1 will have been exchanged.

② (Transpose stage.) At this point, each process possesses a group of h rows of the matrix A, i.e. each process p holds the data that constitutes rows $[ph, ph + h)$, but this data is scrambled in a "block-based" format. This happens because the data exchange in the previous stage was performed in-place. Consequently, the h rows that each process possesses are stored as follows: first sub-column 0, then sub-column w, then sub-column $2w$, and so on until sub-column $(P-1)w$; then sub-column 1, sub-column $w+1$, and so forth until sub-column $(P-1)w+1$; then sub-column 2, and so on. More precisely, let

$$A^{(p)}_{:,k} = \begin{bmatrix} A_{ph,k} \\ A_{ph+1,k} \\ \vdots \\ A_{(p+1)h-1,k} \end{bmatrix},$$

i.e. $A^{(p)}_{:,k}$ is the length h portion of the k-th column of A stored by process p. At the beginning of this stage, each process p has the following elements of A in its memory (read top-to-bottom, left-to-right):

$$\begin{bmatrix} A^{(p)}_{:,0} & A^{(p)}_{:,1} & \cdots & A^{(p)}_{:,w-1} \\ A^{(p)}_{:,w} & A^{(p)}_{:,w+1} & \cdots & A^{(p)}_{:,2w-1} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(p)}_{:,(P-1)w} & A^{(p)}_{:,(P-1)w+1} & \cdots & A^{(p)}_{:,W-1} \end{bmatrix}. \quad (17)$$

However, for the row transforms in the next stage of the computation we require the format

$$\begin{bmatrix} A^{(p)}_{:,0} & A^{(p)}_{:,1} & \cdots & A^{(p)}_{:,W-1} \end{bmatrix}, \quad (18)$$

so that two consecutive sub-columns are contiguous in memory. This corresponds to the region marked by ③ in Fig. 1.

Going from (17) to (18) can be seen as transposing the layout (17), interpreting it as matrix with dimensions $P \times w$, in which each "cell" encompasses h elements (the $A^{(p)}_{:,k}$). This transposition has to be performed in-place, since in general we cannot afford to replicated the portions of A held by the processes. There are several well-known methods for in-place matrix transposition, most of which are based on "following cycles" [15]. These methods work by finding an element that has not yet been visited, freeing it (whilst copying it to somewhere else), determining which element goes into the place that became vacant, moving that element there, and repeating until the element that goes into a newly vacated place is the one that was initially freed. The process is repeated until all elements of the matrix have been visited. These algorithms usually exhibit poor memory locality due to the somewhat arbitrary memory access patterns they exhibit, but since each of our "cells" encompasses h elements, in our case this problem does not apply (we pay a penalty for moving the first element of each group of h, but the remaining $h-1$ elements are moved sequentially). Moreover, we can use a single thread to "swiftly" follow the cycles, moving only the first element of each group of h elements, and defer the bulk of the work, the movement of the remaining $h-1$ elements, to other threads—which allows us to parallelise the operation. Pseudocode for this algorithm is provided in Alg. 5.

③ (Row stage.) At the beggining of this stage, each process stores a "slice" of h rows of matrix A in the layout given in (18), which corresponds to the region marked by ③ in Fig. 1. All that is left to do is to compute the NTT of each of these rows, which can be done for all h rows of each process at once using the Gentleman-Sande algorithm given in Alg. 2.

The steps above compute the forward NTT of A. To compute the inverse transform, the steps must be followed "in reverse" and with the following modifications:

- Use the Cooley-Tukey algorithm given in Alg. 2 to perform denormalised inverse transforms in stage ③.
- In stage ②, instead of going from layout (17) to (18) we want to do the opposite (go from (18) to (17)). This can be achieved by transposing the portion of A each process holds, interpreting it as a $w \times P$ matrix in which each cell contains h elements.
- In stage ①, the pseudocode is changed to:

  for($pw \leq k < pw + w$) {

  1: (Exchange data.)
  alltoall($A_{:,k}$, h);

  2: (Apply twiddle factors.)
  for($0 \leq j < H$) $A_{jk} \leftarrow A_{jk} g^{jk}$;

  3: (Compute inverse NTT of column.)
  $A_{:,k} \leftarrow \mathrm{NTT}^{-1}(A_{:,k})$; ▷ denormalised, Alg. 1

  }

  4: (Normalise.) Divide each element $A_{j,k}$ by m (mod N). Do not convert m^{-1} to Montgomery form beforehand to combine the normalisation with the conversion of the data from Montgomery to standard representation (see Sec. II-B).
  for($0 \leq j < H$) $A_{jk} \leftarrow m^{-1} A_{jk}$;

  As for the forward transform, in practice this computation proceeds in tranches, but this time the master threads starts by exchanging C columns in an all-to-all fashion amongst the processes, and once the communication finishes the worker threads start processing them. Whilst the worker threads process the first tranche, the master thread starts exchanging the second tranche, and so forth.

IV. COUNTING GOLDBACH PARTITIONS

A pair of primes (p, q) that sum to an even number $n = p+q$ is known as a Goldbach partition (or representation) of n. If we denote the number of such partitions for a given even n by $R(n)$, i.e.

$$R(n) = \#\left\{(p, q) : n = p + q \text{ and } p, q \text{ prime}\right\}, \tag{19}$$

then the Goldbach conjecture states that $R(n) > 0$ for all even $n \geq 4$. In this section we outline our method to compute $R(n)$ for all even n below a large limit, based on a strategy described in [2, p. 492].

Intuitively, the main idea is to take the odd primes $(3, 5, 7, 11, \ldots)$ and count how many times each even number $(6, 8, 10, 12, \ldots)$ comes up as the sum of any two primes of the list (which will correspond precisely to the number of Goldbach representations of those even numbers). To achieve this, let us consider the binary sequence

$$(p_k)_{k\geq 0} = (1, 1, 1, 0, 1, \ldots), \tag{20}$$

where p_k is equal to 1 if $2k + 3$ is prime, and 0 otherwise. Then, the act of "counting how many times each even number comes up as a sum of two odd primes" can be carried out by convolving the sequence (p_k) with itself. If we encode (p_k) into a formal power series

$$P(x) = \sum_{k=0}^{\infty} p_k x^k, \tag{21}$$

the result of the aforementioned convolution is the sequence $(q_k)_{k\geq 0}$ encoded in the coefficients of

$$Q(x) = P^2(x) = \sum_{k=0}^{\infty} q_k x^k, \tag{22}$$

with

$$q_k = \sum_{l=0}^{k} p_l p_{k-l} = R(2k + 6). \tag{23}$$

In practice, we cannot work with these infinite series directly since there is no known tractable closed-form expression for the sequence of primes. Instead, we can break $Q(x)$ into smaller polynomials of fixed-size, and express them as a function of similar-sized sub-polynomials of $P(x)$. In this sense, let

$$P_{i,l}(x) = \sum_{k=il}^{il+l-1} p_k x^k, \quad i \geq 0, \quad l \geq 1 \tag{24}$$

be the i-th "chunk" of length l of $P(x)$. Furthermore, let

$$S_{i,l}(x) = \sum_{j=0}^{i} P_{j,l}(x) P_{i-j,l}(x), \tag{25}$$

and

$$L_{i,l}(x) = S_{i,l}(x) \bmod x^{il+l} \tag{26}$$

$$H_{i,l}(x) = S_{i,l}(x) - L_{i,l}(x). \tag{27}$$

Then,

$$Q_{0,l}(x) = L_{0,l}(x), \tag{28}$$

$$Q_{i,l}(x) = L_{i,l}(x) + H_{i-1,l}(x), \quad i \geq 1, \tag{29}$$

and

$$Q_{i,l}(x) = \sum_{k=il}^{il+l-1} q_k x^k, \tag{30}$$

with q_k as in (23). The idea is to break $Q(x)$ into smaller polynomials $Q_{i,l}(x)$, and express the latter as sums of products of chunks of $P(x)$ of length l. The computation then proceeds as if multiplying two numbers using the schoolbook multiplication algorithm: each sum of products $S_{i,l}(x)$ generates a low and a high part, $L_{i,l}(x)$ and $H_{i,l}(x)$, and each $Q_{i,l}(x)$ is equal to the sum of the low part of the i-th sum with the high part ("the carry") of the previous sum.

A. *Generating elements* p_k

Now we consider how to generate the terms of the sequence (p_k) for the polynomials $P_{i,l}(x)$. Due to the memory layout used for the distributed transforms (see Sec. III-D and Alg. 4), we will consider a more general family of sequences of the form $(s_{k;a,b})_{k\geq 0}$, with $a = 2^d, d \geq 1$, $b \geq 3$ and odd, and where $s_{k;a,b}$ is equal to 1 if $ak + b$ is prime, and 0 otherwise. Clearly, $p_k = s_{k;2,3}$. The algorithm works by sieving through the sequence $(b, a + b, \ldots, (l - 1)a + b)$ in a similar fashion to the segmented sieve of Eratosthenes.

Algorithm 3 (Generate p_k in progression.) Given a, b as defined above and $l \geq 1$, compute $(s_{k;a,b})_{k=0}^{l-1}$. We assume the set $\mathcal{P}$ of primes in $[3, \sqrt{(l-1)a + b}]$ has been precomputed, which can be done using any standard sieving method such as [2, Algorithm 3.2.2] or a precomputed table.

1: (Initialise.)
   for($0 \leq i < l$) $s_i \leftarrow 1$; ▷ or Montgomery of 1, see below

2: (Sieve.)

```
for(p ∈ 𝒫) {
    k ← (−ba⁻¹) mod p;
    if(ak + b = p) k ← k + p;
    while(k < l) {
        s_k ← 0;
        k ← k + p;
    }
}
```

The inverses of a are guaranteed to exist since $\forall_{p\in\mathcal{P}} \gcd(a, p) = 1$. Moreover, in practice a is a constant throughout the whole computation (see steps 6 and 8 of Alg. 4), and so its inverses modulo the primes in $\mathcal{P}$ can be precomputed. Furthermore, since we use Montgomery representation for the modular arithmetic operations, in practice we perform the initialisation above with the Montgomery representation of one instead of the literal unity, which spares us from having to perform the conversion to Montgomery representation explicitly.

B. Computing $R(n)$ in chunks

The polynomial products $P_{j,l}(x)P_{i-j,l}(x)$ in (25) can be seen as the *acyclic* convolution of the coefficients of $P_{j,l}(x)$ and $P_{i-j,l}(x)$. As we saw in Sec. III-A, the *cyclic* convolution of two sequences can be computed by taking their discrete Fourier (or similar) transform, multiplying the transformed elements element-by-element, and taking the inverse transform of the result. Acyclic convolution can be computed in a similar fashion: if we double the length of the sequences being convolved and set the higher-order coefficients of the result to zero, the *cyclic* convolution of the doubled sequences will match the *acyclic* convolution of the original sequence. Since in our case the sequences we are working with are integer sequences and we require all convolutions to be computed exactly (so that the computations of $R(n)$ are exact as well), the best option is to use NTTs to perform these convolutions. To this end, the computation of Goldbach partitions by means of distributed NTT convolutions proceeds as follows.

Algorithm 4 (Compute $R(n)$ in chunks.) Given l and m, with $l = 2^d, d \geq 0$, and $m \bmod l = 0$, determine the number of Goldbach partitions of the even numbers in $[6, 2m+4]$, l terms at a time. This algorithm requires three buffers A, B, and C, each of length $2l$. It uses the algorithm of Sec. III-D to compute distributed length-$2l$ NTTs.

1: Let $2l = H \times W$. Buffers A, B, and C are viewed as matrices $H \times W$ that correspond to sequences a, b and c. These sequences are organised conceptually in row-major order in the matrices, i.e. $A_{j,k}$ corresponds to a_{jW+k} (similarly for b and c), but are stored (physically, in memory) in column-major order, so that elements a_i and a_{i+1} actually reside H elements apart (similarly for b and c). In practice, A, B, and C are distributed across several processes, with each process initially storing a group of consecutive columns (see Sec. III-D). Finally, let $A_{i:j,k}$ denote the range elements $A_{i,k}, A_{i+1,k}, \ldots, A_{j-1,k}$.

2: (Initialise.) Set initial carry to zero. Corresponds to zeroing the l highest terms of c.

for($0 \leq k < W$) $C_{H/2:H,k} \leftarrow 0$;

3: (Compute each $Q_{i,l}(x)$.)

for($0 \leq i < m/l$) {

4: (Move carry out to carry in and transform it.)

for($0 \leq k < W$) {
  $C_{0:H/2,k} \leftarrow C_{H/2:H,k}$;
  $C_{H/2:H,k} \leftarrow 0$;
}
$C \leftarrow \mathrm{NTT}(C)$;

5: (Compute $S_{i,l}(x)$.)

for($0 \leq j < \lceil i/2 \rceil$) {

6: (Prepare inputs.) a gets the coefficients of $P_{j,l}(x)$ computed in a column-based arithmetic progression via Alg. 3, and similarly for b (which gets the coefficients of $P_{i-j,l}(x)$). Additionally, the sequences are padded with l zeros to prepare for the acyclic convolution to be computed.

for($0 \leq k < W$) {
  $A_{0:H/2,k} \leftarrow (s_{n;2W,2(jl+k)+3})_{n=0}^{H/2-1}$;
  $A_{H/2:H,k} \leftarrow 0$;
  $B_{0:H/2,k} \leftarrow (s_{n;2W,2((i-j)l+k)+3})_{n=0}^{H/2-1}$;
  $B_{H/2:H,k} \leftarrow 0$;
}

7: (Polynomial product.) Compute $P_{j,l}(x)P_{i-j,l}(x)$ via "incomplete" convolution (i.e., for the time being, without inverse transform) and accumulate.

$C \leftarrow C + 2 * \mathrm{NTT}(A) * \mathrm{NTT}(B)$;
}

8: (If i is even, accumulate $P^2_{i/2,l}(x)$.)

if($i \bmod 2 = 0$) {
  for($0 \leq k < W$) {
    $A_{0:H/2,k} \leftarrow (s_{n;2W,il+2k+3})_{n=0}^{H/2-1}$;
    $A_{H/2:H,k} \leftarrow 0$;
  }
  $C \leftarrow C + \mathrm{NTT}(A)^2$; ▷ Element-wise square
}

9: (Transform back.) This completes the convolutions.

$C \leftarrow \mathrm{NTT}^{-1}(C)$;

10: (Process $R(n)$.) At this point, we have

$$C_{j,k} = c_{jW+k} = R(2(il + jW + k) + 6),$$

with $0 \leq j < H/2$ and $0 \leq k < W$.

In the algorithm above we make use of the symmetry of the sum in (25) to reduce the number of polynomial products required to evaluate $S_{i,l}(x)$ from $i+1$ to $\lfloor (i+2)/2 \rfloor$. Furthermore, due to the linearity property of the DFT, we avoid computing one inverse transform per polynomial product and, instead, accumulate the polynomial products in the transformed space and compute a single inverse transform once all polynomials have been accumulated.

V. Results and evaluation

In this section we present the main results of our work. We list the compiler versions and optimisation flags utilised for the compiler comparison of Sec. V-A in Tab. I. For all other tests we used GCC 10.2 with the flags given in the same table.

Table I
Compiler versions and optimisation flags.

Compiler	Flags
Arm 21.0	-O3 -march=armv8.2-a+sve -mcpu=a64fx -fopenmp
Cray 10.0.1	-O3 -h aggress,flex_mp=tolerant,vector3,omp
Fujitsu 4.3.1	-O3 -KA64FX,SVE,fast,preex,openmp,simd=auto
GCC 10.2 & 11	-O3 -march=native -mcpu=native -fopenmp

For the tests involving MPI and OpenMP, we have placed one MPI process per NUMA domain (or Core Memory Group in the case of the A64FX), and launched as many OpenMP

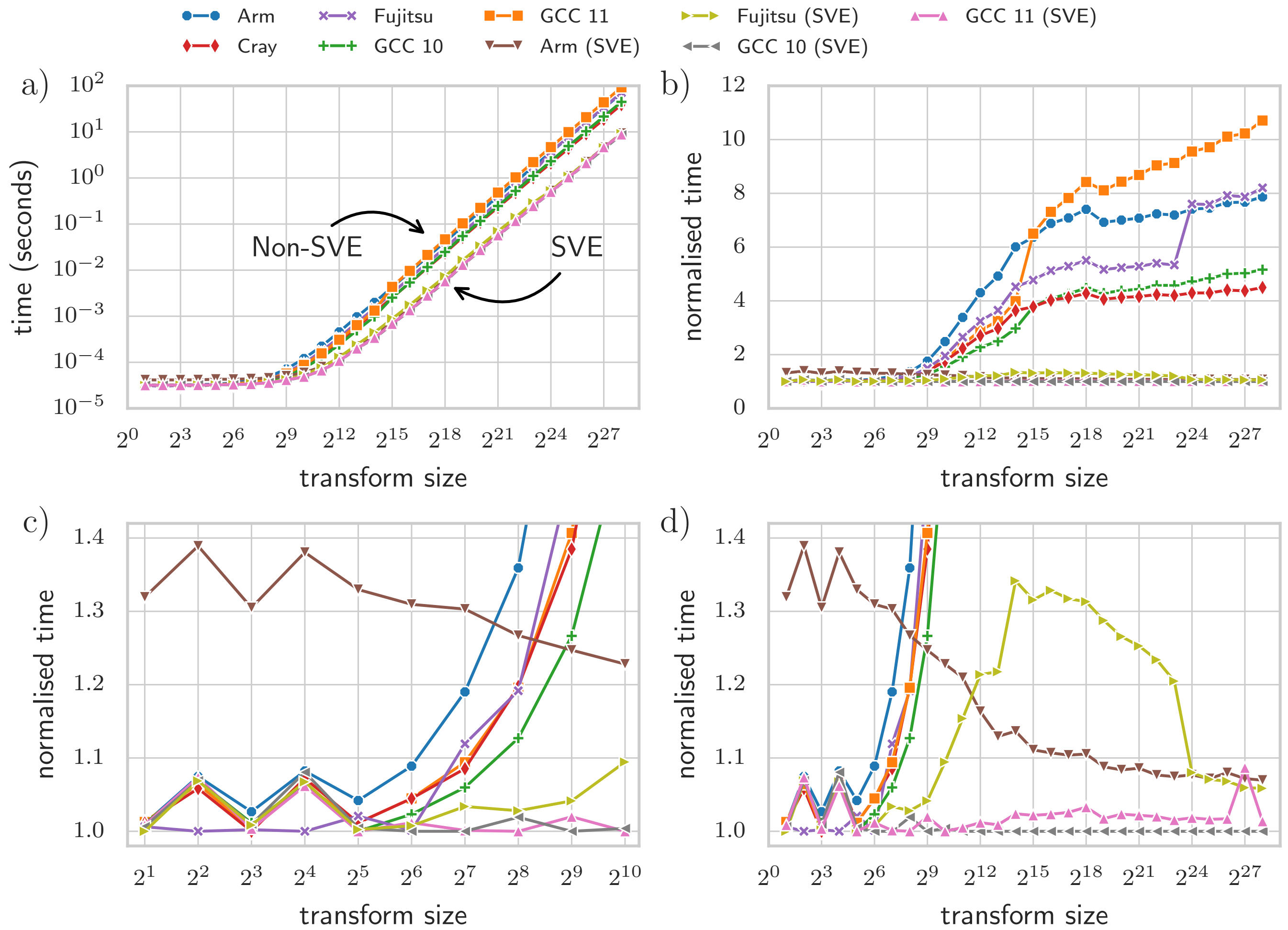

Figure 2. Impact of using the SVE routines of Sec. II to implement a Stockham NTT. (a) average runtime per transform size for the compilers tested (number of runs varies between 1000 for the smaller transforms and 10 for the larger ones); (b) average runtime normalised to the performance of the fastest compiler per transform size (lower is better); (c) zoom in view of small transform sizes; (d) zoom in view of SVE-based implementations.

threads per MPI process as the number of cores in the NUMA domain. All threads were pinned to cores. As mentioned in Sec. III-D, only the master thread issues MPI calls. On Fulhame the MPI library used was OpenMPI 4.0.2, on Isambard 1 Cray MPICH 7.7.17, and on Isambard 2 OpenMPI 4.1.0.

A. *Local transforms with SVE*

In Fig. 2 we evaluate the impact of using the SVE-based modular arithmetic routines described in Sec. II to vectorise the innermost loop of a Stockham NTT (Alg. 1), using the four major Arm compilers (Arm, Cray, Fujitsu, and Gnu) to build the code. These tests were run on an A64FX core on Isambard 2. Since the other two local NTT routines employed in this work are used to compute multiple NTTs all at once, they exhibit a similar inner loop structure to Stockham's NTT, and thus these results also apply to them.

In Fig. 2-a we plot the average time taken to perform transforms of varying sizes. As the panel shows, the SVE routines improve the runtime of the computation considerably. To show the difference in performance between the compilers and SVE/non-SVE versions more clearly, in Fig. 2-b we have normalised the runtime of each compiler and version, for each transform size, to the time obtained by the fastest compiler/version for that size, so that the fastest configuration has a normalised time of 1. This panel shows that, for mid/long-sized transforms (over 2^{14} points), the SVE versions achieve a speedup of more than 4x over the fastest compilers using the non-SVE versions of the modular arithmetic routines. Moreover, GCC 10.2 and 11 exhibit a significant difference in performance when using the non-SVE versions of the modular arithmetic routines—the former being approximately two times faster than the latter for mid/long-sized transforms. This seems to be due to the cost model of the A64FX processor which was introduced in GCC 11, although this requires further investigation.

In Fig. 2-c we show a zoomed in view of Fig. 2-b for very short transform sizes. It demonstrates that even for extremely small sizes, the SVE versions still remain competitive with

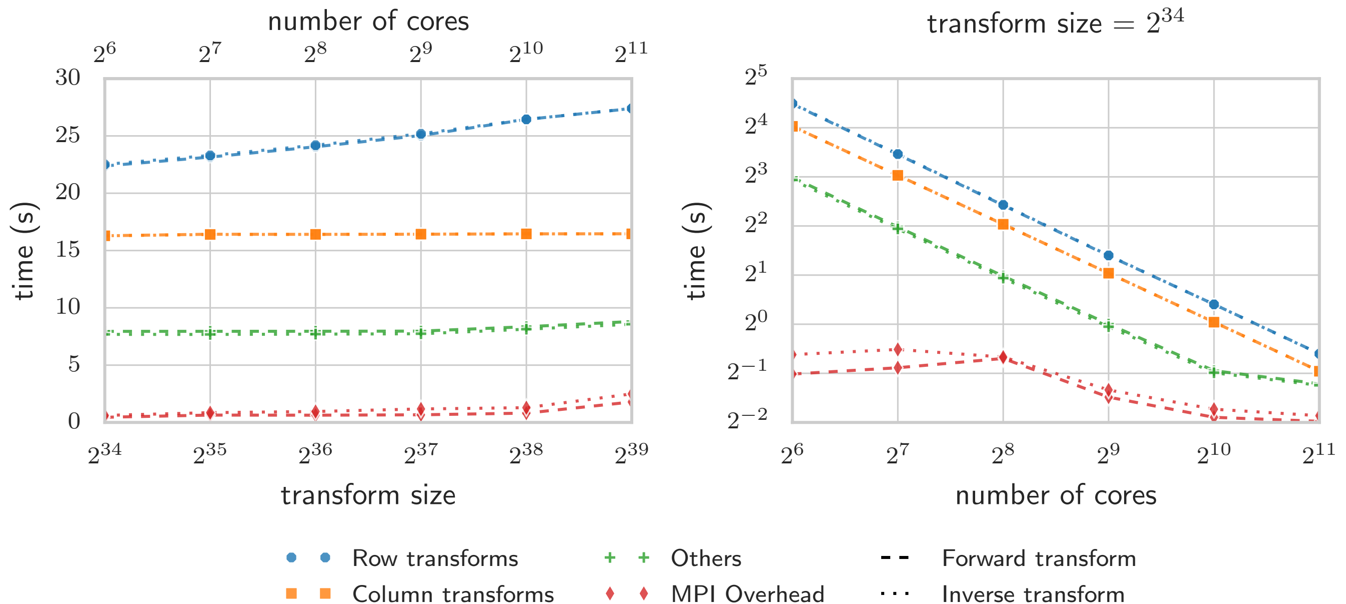

Figure 3. Breakdown of the execution of forward and inverse distributed transforms (average of 10 runs). Using up to 32 HPE Apollo 70 nodes on Fulhame, 1 MPI process per socket, 32 OpenMP threads per MPI process. Left: Weak scaling. Right: Strong scaling.

the non-SVE versions. Finally, in Fig. 2-d we show another zoomed in view of Fig. 2, but this time focused on the SVE versions (over the complete range of transform sizes considered). Since we use SVE intrinsics to implement the SVE modular arithmetic routines, we expected the compilers to have little impact in the performance of the transforms (because the compilers are essentially limited to reorder the instructions). Whilst it is true that the differences amongst the SVE versions are far less pronounced than those amongst non-SVE versions, there are still a couple of interesting patterns to observe. First, in this experiment the fastest compiler turned out to be GCC in both versions 10.2 and 11, with the former being slightly faster than the latter. Secondly, the Arm compiler starts off being around 30–40% slower than the fastest compiler (GCC 10.2), but this figure is reduced to slightly below 10% as the transform sizes increase. Lastly, the Fujitsu compiler seems to perform comparatively poorly with respect to the fastest compiler for transform sizes between 2^{10} and 2^{23}. The reason for this poorer performance is currently not understood.

B. Distributed transforms

In Fig. 3 we show a breakdown of the time spent on the major steps of the computation of distributed NTTs (Sec. III). For all transforms computed, we took $H = 2^{16}$ and $W = m/H$, where m is the transform size. These tests were run on Fulhame, and thus they used the non-SVE versions of the modular arithmetic routines. “Row/column transforms” corresponds to the time spent computing local row/column NTTs (stages ③ and ① of Sec. III-D, respectively). “Others” corresponds to intermediate computations, such as applying twiddle factors in stage ① and performing the in-place transpose of stage ②. “MPI Overhead” corresponds to the “dead time” of the computation during which threads wait for MPI communication and/or synchronisation during the exchange of data of stage ① (i.e. the time spent in MPI that is not overlapped with computation).

The plots show that our methods are scaling almost perfectly, especially in the regime of weak scaling (left panel). In this regime, since H is held constant for all transform sizes, ideally the time spent on all steps except “row transforms” would remain constant, and “row transforms” would increase linearly with the logarithm of the transform size (due to the logarithm factor in the $O(n \log n)$ asymptotic complexity of NTTs), which would show up as a straight line in the plot Indeed, we can see this is almost exactly what is happening. Moreover, the wait time in MPI (the MPI overhead) remains below 4% throughout the whole range of transform sizes tested, which demonstrates that we have successfully managed to overlap the MPI communications with the computations. Our methods also exhibit good performance in the regime of strong scaling (right panel), although in this case the MPI overhead starts to take over a bigger percentage of the computation time as the number of cores increases. This is somewhat expected, since by holding H fixed while the number of MPI processes increases, the number of columns per MPI process is reduced to very low numbers that do not allow for effective overlapping. A better strategy for improved strong scaling would be to decrease H (or, equivalently, increase W) as the number of MPI processes increases, so that the number of columns per MPI process is kept at a relatively large number. However, to be most effective, this

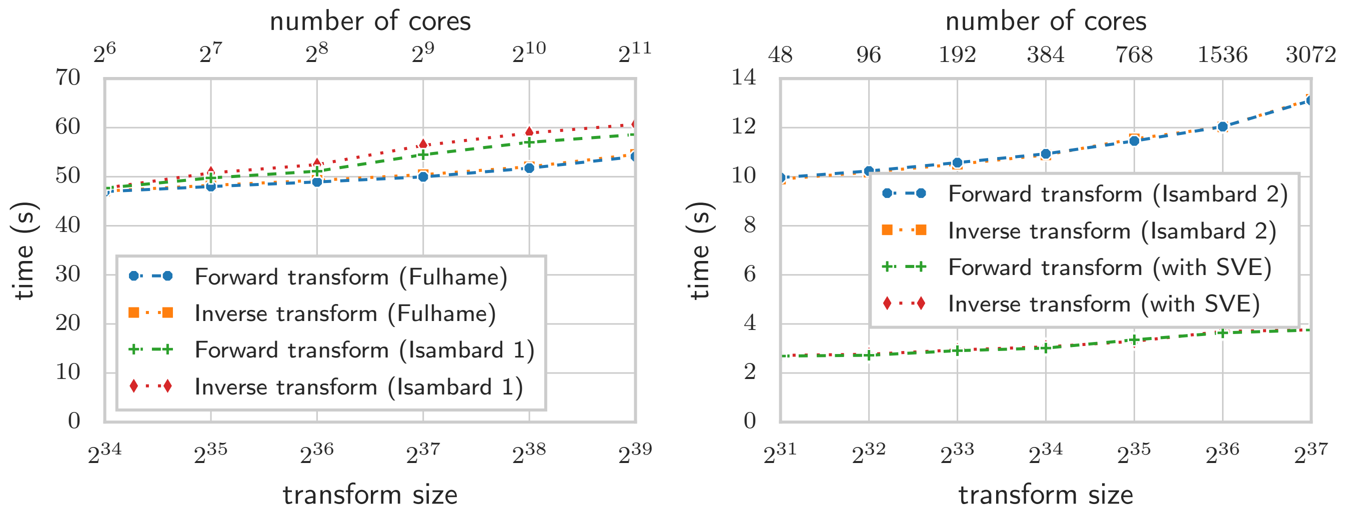

Figure 4. Left: Comparison of Fulhame (HPE Apollo 70) and Isambard 1 (Cray XC50) systems. Right: Impact of SVE routines in the distributed NTT routines, evaluated on Isambard 2 (HPE Apollo 80). Averages of 10 runs.

strategy should be combined with a different algorithm for the transpose in stage ② of Sec. III-D, as the algorithm we have used works best when the number of *rows* per MPI process is relatively large.

In Fig. 4 we show results of our distributed transforms on other systems. On the left panel, we compare the results obtained on Fulhame with those obtained on Isambard 1. This is not a completely fair comparison, since Isambard 1 was experiencing a much higher workload than Fulhame at the time we conducted our experiment (in fact, on Fulhame, we were at times able to run our tests in isolation). Nevertheless, Isambard 1 still achieved results within 10% of those obtained on Fulhame. Meanwhile, on the right panel, we compare the performance of the distributed NTTs, with and without the SVE-based modular arithmetic methods, on Isambard 2. As the plot shows, the SVE-based methods provided a speedup of more than 3.5x over the non-SVE routines.

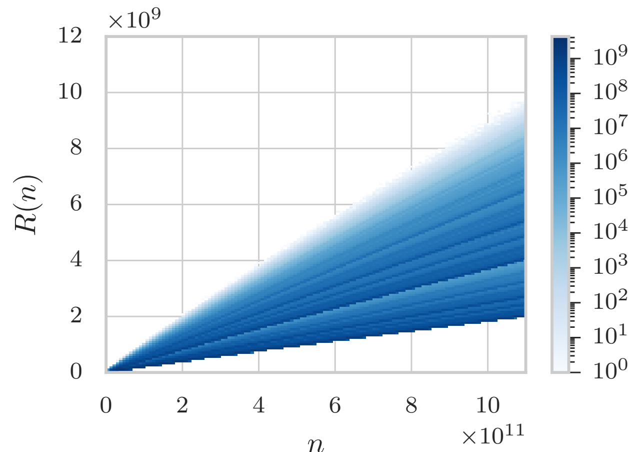

Figure 5. Histogram in a 128×128 grid of the number of Goldbach partitions of the even numbers up to 2^{40}.

C. Counting Goldbach partitions

In Fig. 5 we show a histogram of the number of Goldbach partitions $R(n)$ of all even numbers in $[4, 2^{40}]$. These are early results in preparation for a larger scale computation we wish to undertake in the future to reach 2^{45}. Reaching 2^{40}, already 2000x more than the previously published record [4], took just under 10 minutes compute time using 32 nodes of Fulhame (2048 cores in total), utilising intermediate distributed transforms of 2^{38} points each. The correctness of the results was independently verified in intervals of the form $[k2^{38} - 126, k2^{38}]$, $k = 1, 2, 3, 4$ by computing the number of Goldbach partitions in those intervals directly.

VI. Related work

Modular arithmetic is a crucial component of many algorithms for computer algebra, cryptography, and several other areas. The most important operation to optimise is multiplication modulo a fixed integer, as this is the building block of other important nontrivial modular operations (e.g. exponentiation). The most commonly used methods for performing fast modular multiplications are based on Knuth's long divide algorithm [8, Algorithm 4.3.1D] and on the Barrett [9] and Montgomery methods [1]. A survey on these and other modular multiplication strategies is given in [16]. A comparison of them is also given in [10].

Whilst there has been extensive research on the optimisation of modular multiplications for GPUs [17]–[20], efforts to optimise modular operations for SIMD instruction sets have been somewhat more scarce. Alongside their work modular arithmetic optimisation for GPUs, in [17] the authors also consider optimisations for several x86 CPUs supporting SSE2

(Streaming SIMD Extensions 2). A more extensive work on modular arithmetic optimisation for x86 SIMD extensions has been carried in [21], where the authors develop efficient implementations of several modular operations for the SSE and AVX instruction sets (in particular SSE4.2 and AVX2 for the Barett and Montgomery methods). Taking the previous work as a stepping stone, in [22] the authors implement an efficient modular multiplication algorithm for AVX-512. To the best of our knowledge, there has been no similar work considering the implementation of efficient modular arithmetic methods for Arm SVE.

Several optimisations for NTTs have also been proposed. Some recent examples include an alternative modular reduction step that is more appropriate for relatively small moduli (e.g. below 2^{32}) [23], and specialised butterflies that avoid some intermediate adjustment steps [24]. NTT implementations on GPUs have also been considered [25].

Finally, to the best of our knowledge the largest published computation of all Goldbach partitions to a large limit was carried out by Richstein [4], who also utilised a method based on the convolution of the sequence of odd primes, although implemented in a different way.

VII. Conclusions

We have successfully shown that SVE can be utilised to efficiently vectorise non-trivial modular arithmetic loops. By implementing modular addition, subtraction and multiplication with SVE intrinsics, we have achieved speedups of more than 4 in number-theoretic transform codes. Moreover, we have evaluated how efficiently the four major Arm compilers (Arm, Cray, Fujitsu and Gnu) compile code involving modular arithmetic, where we have observed a large discrepancy between compilers. Namely, when using the non-SVE routines for modular arithmetic, GCC 10.2 is twice as fast as GCC 11.

Furthermore, we have successfully implemented distributed number-theoretic transform methods that achieve nearly complete overlap of MPI communication with computation. Additionally, by combining local NTT routines with different characteristics, and working with a matrix-like layout where the data is arranged (conceptually) by row, but stored (physically) by column, we have been able to implement a very efficient in-place parallel transpose algorithm. The result of combining good overlapping of communication with computation, and fast in-place transpose, is that our distributed transforms show very good scalability to thousands of cores. For example, the MPI communications' wait time (i.e. time not overlapped with computations) in a transform of 2^{39} points using 32 ThunderX2-based nodes of Fulhame (2048 cores in total) is less than 4% of the total running time of the transform, and the time spent in the in-place transpose is a small fraction (less than 5%) of the whole computation.

Ultimately, the distributed NTTs we have developed enabled us to compute the number of Goldbach partitions of all even numbers not greater than 2^{40} via exact integer convolutions. This corresponds to a 2000x improvement over previously published results. In the near future, we plan to extend this computation further up to 2^{45}, to study several properties of these numbers.

Acknowledgments

The Fulhame HPE Apollo 70 system is supplied to EPCC, the supercomputing centre at the University of Edinburgh, as part of the Catalyst UK programme, a collaboration with Hewlett Packard Enterprise, Arm and SUSE to accelerate the adoption of Arm based supercomputer applications in the UK.

This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1).

References

[1] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–519, May 1985.

[2] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, 2nd ed. New York: Springer-Verlag, 2005.

[3] D. H. Bailey, "FFTs in external or hierarchical memory," *The Journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, Mar. 1990.

[4] J. Richstein, "Computing the Number of Goldbach Partitions up to 5 108," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, W. Bosma, Ed. Berlin, Heidelberg: Springer, 2000, pp. 475–490.

[5] "Arm C Language Extensions for SVE," Tech. Rep. 00bet6.

[6] "A64FX Microarchitecture Manual," accessed: 2021-01-26. [Online]. Available: https://github.com/fujitsu/A64FX

[7] J. Arndt, *Matters Computational: Ideas, Algorithms, Source Code*. Berlin Heidelberg: Springer-Verlag, 2011.

[8] D. E. Knuth, *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor," in *Advances in Cryptology — CRYPTO' 86*, ser. Lecture Notes in Computer Science, A. M. Odlyzko, Ed. Berlin, Heidelberg: Springer, 1987, pp. 311–323.

[10] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Advances in Cryptology — CRYPTO' 93*, ser. Lecture Notes in Computer Science, D. R. Stinson, Ed. Berlin, Heidelberg: Springer, 1994, pp. 175–186.

[11] E. De Win, S. Mister, B. Preneel, and M. Wiener, "On the performance of signature schemes based on elliptic curves," in *Algorithmic Number Theory*, ser. Lecture Notes in Computer Science, J. P. Buhler, Ed. Berlin, Heidelberg: Springer, 1998, pp. 252–266.

[12] W. Cochran, J. Cooley, D. Favin, H. Helms, R. Kaenel, W. Lang, G. Maling, D. Nelson, C. Rader, and P. Welch, "What is the fast Fourier transform?" *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, Oct. 1967, conference Name: Proceedings of the IEEE.

[13] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Jan. 1992.

[14] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[15] F. G. Gustavson and D. W. Walker, "Algorithms for in-place matrix transposition," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 13, 2019.

[16] N. Nedjah and L. de Macedo Mourelle, "A review of modular multiplication methods and respective hardware implementation," *Informatica*, vol. 30, no. 1, 2006.

[17] D. Bernstein, H.-C. Chen, M. Chen, C. Cheng, C. Hsiao, T. Lange, Z. Lin, and B. Yang, "The billion-mulmod-per-second PC," in *SHARCS'09 workshop record (proceedings 4th workshop on special-purpose hardware for attacking cryptograhic systems, lausanne, switserland, september 9-10, 2009)*, 2009, pp. 131–144.

[18] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUs for RSA," in *Information security*, S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. M. Yiu, Eds. Cham: Springer International Publishing, 2014, pp. 198–215.

[19] S. Antão, J.-C. Bajard, and L. Sousa, "Elliptic Curve point multiplication on GPUs," in *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*. Rennes, France: IEEE, Jul. 2010, pp. 192–199.

[20] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of modular arithmetic algorithms on GPUs," in *PARCO*, ser. Advances in parallel computing, B. M. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. J. Peters, and T. Priol, Eds., vol. 19. IOS Press, 2009, pp. 315–322.

[21] J. V. D. Hoeven, G. Lecerf, and G. Quintin, "Modular SIMD arithmetic in Mathemagix," *ACM Transactions on Mathematical Software*, vol. 43, no. 1, pp. 1–37, Aug. 2016.

[22] P. Fortin, A. Fleury, F. Lemaire, and M. Monagan, "High-performance SIMD modular arithmetic for polynomial evaluation," *Concurrency and Computation: Practice and Experience*, May 2021.

[23] P. Longa and M. Naehrig, "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography," in *Cryptology and Network Security*, S. Foresti and G. Persiano, Eds. Cham: Springer International Publishing, 2016, vol. 10052, pp. 124–139, series Title: Lecture Notes in Computer Science.

[24] D. Harvey, "Faster arithmetic for number-theoretic transforms," *Journal of Symbolic Computation*, vol. 60, pp. 113–119, Jan. 2014.

[25] W.-K. Lee, S. Akleylek, D. C.-K. Wong, W.-S. Yap, B.-M. Goi, and S.-O. Hwang, "Parallel implementation of Nussbaumer algorithm and number theoretic transform on a GPU platform: application to qTESLA," *The Journal of Supercomputing*, vol. 77, no. 4, pp. 3289–3314, Apr. 2021.

APPENDIX A
PSEUDOCODE FOR IN-PLACE PARALLEL TRANSPOSE

Algorithm 5 (In-place parallel transpose.) Given an array X that stores in column-major order a matrix $m \times n$ where each "cell" is r elements long (see (17) for an example), transpose X in-place. We assume all numbers are stored modulo N (we use this property to temporarily mark visited elements). We utilise OpenMP to illustrate how to parallelise the algorithm. Finally, we assume each thread has an auxiliary buffer T of length r.

1: (Initialise.)

▷ A single thread follows the cycles ''swiftly''
#pragma omp single
for($k \leftarrow 0$, $k' \leftarrow 0$; $k < mn$; $k \leftarrow k' + 1$) {

2: (Find next cycle.)

while($X_{kr} \geq N$) $k \leftarrow k + 1$;
$k' \leftarrow (k \bmod m)n + \lfloor k/m \rfloor$;
if($k \neq k'$) {

3: (Follow the cycle that starts at k.)

$k_0 \leftarrow k$;
▷ Task thread moves elements $[1, r)$
#pragma omp task
{
for($0 < j < r$) $T_j \leftarrow X_{kr+j}$;
do {
for($0 < j < r$) $X_{kr+j} \leftarrow X_{k'r+j}$;
$k \leftarrow k'$;
$k' \leftarrow (k \bmod m)n + \lfloor k/m \rfloor$;
} while($k' \neq k_0$);
for($0 < j < r$) $X_{kr+j} \leftarrow T_j$;
}
▷ Main thread moves only element $[0]$
{
$t \leftarrow X_{kr}$;
do {
$X_{kr} \leftarrow X_{k'r} + N$;
$k \leftarrow k'$;
$k' \leftarrow (k \bmod m)n + \lfloor k/m \rfloor$;
} while($k' \neq k_0$);
$X_{kr} \leftarrow t$;
}
}

4: (Mark cycle k as followed.)

$X_{kr} \leftarrow X_{kr} + N$;
}

5: (Clear all marks.)

#pragma omp for schedule(static)
for($0 \leq k < mn$) $X_{kr} \leftarrow X_{kr} - N$;