AVX-512-based Parallelization of Block Sieving and Bucket Sieving for the General Number Field Sieve Method*

Pritam Pallab and Abhijit Das
Indian Institute of Technology, Kharagpur, India

Keywords: General Number Field Sieve Method, RSA Cryptanalysis, Line Sieving, Lattice Sieving, Block Sieving, Bucket Sieving, Single Instruction Multiple Data (SIMD), Multi-core, Multi-thread, AVX-512, Skylake.

Abstract: The fastest known general-purpose technique for factoring integers is the General Number Field Sieve Method (GNFSM), in which the most time-consuming part is the sieving stage. For both line sieving and lattice sieving, two cache-friendly extensions used in practical implementations are block sieving and bucket sieving. The new AVX-512 instruction set in modern Intel CPUs offers some fast vectorization intrinsics. In this paper, we report our AVX-512 based cache-friendly parallelization of block and bucket sieving for the GNFSM. We use vectorization for both sieve-index calculations and sieve-array updates in block sieving, and for the insertion stage in bucket sieving. Our experiments using Intel Xeon Skylake processors demonstrate a performance boost in both single-core and multi-core environments. The introduction of cache-friendly sieving leads to a speedup of up to 63%. On top of that, vectorization yields a speedup of up to 25%.

1 INTRODUCTION

The General Number Field Sieve Method (GNFSM) (Lenstra et al., 1993a) is the fastest known technique for factoring large composite integers, like RSA moduli. The RSA (Rivest–Shamir–Adleman) algorithm (Rivest et al., 1978) is one of the earliest and most widely used public-key cryptographic algorithms, and exploits the difficulty of factoring products of pairs of large primes to derive its security. The last published successful RSA factorization attempt was that of an RSA modulus of length 795 bits (Boudot et al., 2020a). There is also an unpublished claim of successful factorization of RSA-250 (Boudot et al., 2020b) which is a 829-bit RSA modulus. All these attempts implement the GNFSM.

The GNFSM originates from a specialized form called the Special Number Field Sieve Method (SNFSM) (Lenstra et al., 1990) which is developed to factor composite integers of the form $r^e \pm s$, where $r, s, e \in \mathbb{Z}$ and $e > 0$. It is asymptotically faster than the GNFSM, and is used to factor the ninth Fermat Number $F_9 = 2^{512} + 1$ (Lenstra et al., 1993b). The SNFSM is later generalized to the GNFSM to work for any composite integer (Buhler et al., 1993). This method is based on a ring homomorphism (Briggs, 1998) $\mathbb{Z}[\theta] \rightarrow \mathbb{Z}$ for a suitable algebraic number θ, and is intended to discover a non-trivial Fermat congruence of the form $x^2 \equiv y^2 \pmod{n}$.

The GNFSM consists of multiple stages, among which sieving is the most time-consuming one taking around 60–80% of the overall running time. There are two main techniques used for sieving: line sieving and lattice sieving. In this paper, we mainly focus on line sieving. In both of these types of sieving, memory accessing plays a pivotal role. In order to minimize costly cache misses, two new modifications are introduced. These are called block sieving (Wambach and Wettig, 1995) and bucket sieving (Aoki and Ueda, 2004). In the recent factorization attempts, the block and bucket sieving ideas are extensively used. Earlier, SSE2- and AVX-based SIMD parallelization techniques are attempted (Sengupta and Das, 2017) for line and lattice (Pollard, 1993) sieving. In that work, the index-calculation part is vectorized, but the sieve-array updating part is not. This is attributed to limited and costly intrinsics available in previous generations of CPUs. The recent introduction of AVX-512 offers a new set of intrinsics, and thereby opens the opportunities of exploring the potentials of fully vectorizing the sieving stage. In this paper, we report our AVX-512-based vectorization attempts for cache-friendly block- and bucket-sieving variants of line sieving. We are able to achieve speedup factors of up to 63% with cache-friendly sieving and additional speedup factors of up to 25% with vectorization.

The rest of the paper is organized as follows. Section 2 deals with the background and a study of ex-

*Funded partially by the Ministry of Electronics and Information Technology, India.

Pallab, P. and Das, A.
AVX-512-based Parallelization of Block Sieving and Bucket Sieving for the General Number Field Sieve Method.
DOI: 10.5220/0010515206530658
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 653-658
ISBN: 978-989-758-524-1

isting factoring algorithms. Section 3 elaborates our vectorization approaches for both block sieving and bucket sieving. The experimental results for single-core and multi-core environments are presented in Section 4. Section 5 concludes the paper with notes on possible extensions of our current work.

2 BACKGROUND

2.1 General Number Field Sieve Method

In order to factor n, the GNFSM starts with the selection of two irreducible polynomials $f_r(x)$ and $f_a(x)$ of degrees d_r and d_a and with a common root m modulo n. Here, f_r (with $d_r = 1$) pertains to the rational side, whereas f_a (with $d_a > 1$) pertains to the algebraic side. We also let $\theta \in \mathbb{C}$ be a root of $f_a(x)$. Next, the rational (RFB) and the algebraic (AFB) factor bases are created. RFB consists of small (integer) primes bounded by a limit B_r, while the AFB consists of prime ideals in the number ring $\mathbb{Q}[X]/\langle f_a(X)\rangle$ of prime norms bounded by a limit B_a. For each small prime p, the prime ideals of norm p can be obtained by identifying the roots of the equation $f_a(x)$ modulo p, that is, by solving the congruence $f_a(r) \equiv 0 \pmod{p}$.

The sieving stage uses two integer parameters a and b with $\gcd(a,b) = 1$. If $a + bm$ and $a + b\theta$ are both smooth over the respective factor bases, a relation is discovered. The integer $a + bm$ is called smooth if it factors completely over the RFB, whereas the algebraic number $a + b\theta$ is called smooth if the ideal $\langle a + b\theta\rangle$ factors completely over the prime ideals in AFB. A choice of the pair (a,b) gives a relation if and only if both the integers $(-b)^{d_r} f_r(-a/b)$ and $(-b)^{d_a} f_a(-a/b)$ factor completely over the primes $\leq B_r$ and B_a. Using the ring homomorphism $\eta : \mathbb{Z}[\theta] \to \mathbb{Z}/n\mathbb{Z}$ taking $\theta \mapsto m$, each relation $\eta(a + b\theta) \equiv a + bm \pmod{n}$ gives a linear congruence. The resulting linear system is solved to reach the Fermat congruence. If $\gcd(x - y, n)$ is a trivial factor of n, we go for the other solutions else we report the factors.

2.2 Sieving

The main focus of this paper is on the efficient implementations of the sieving part mainly the line sieving whereas the proposed methods are applicable to lattice sieving in a straightforward manner. Block and bucket sieving techniques are the cache-friendly extensions of normal sieving.

2.2.1 Block Sieving

Instead of accessing the whole sieve array S for each factor f in the factor base FB, we divide S into multiple blocks and perform sieving on one block at a time. The entire sieve line of length $2MAX_A + 1$ is subdivided into b_n blocks, where the size of each block is $b_s = \lceil (2MAX_A + 1)/b_n \rceil$. This method is advantageous if we keep the value of b_s within the size of the available cache memory. This enables the runtime system to load a whole block of S at a time in the cache, and for all $f \in FB$, accesses are made within that block only. This reduces the cache misses significantly, thereby speeding up the whole sieving process. The factor base is subdivided into two parts: $0 \leq f \leq FB_S$ and $FB_S < f \leq FB_{MAX}$. For the smaller factors ($f \leq FB_S$), block sieving is used.

2.2.2 Bucket Sieving

In order to manage the cache memory efficiently for large factors also, bucket sieving is introduced. Instead of performing normal sieving over large factors, buckets are created, filled, and sieved only one at a time. The main concept relies again on the use of only a portion of the sieve array S during accessing and log subtractions. Let B_n be the number of buckets under consideration, and $B_s = \lceil (2MAX_A + 1)/B_n \rceil$. Bucket sieving employs a two-fold approach. For each large factor f and for each of the sieving location a we encounter, an element $(a, \log(f))$ is inserted in the $\lfloor (a + MAX_A)/B_s \rfloor$-th bucket. The size of each bucket is kept within the limits of the available cache memory. Later, we iterate over all the buckets one by one, popping its elements and performing S updates accordingly. As each of the buckets holds the elements having a within the range of the cache size, cache misses are reduced drastically.

3 OUR IMPLEMENTATION APPROACH

In this section, we elaborate our approach of using SIMD (Single Instruction Multiple Data) in the context of block sieving and bucket sieving. The latest SIMD feature added by Intel is AVX-512 which supports 512-bit registers. In the context of sieving, it allows us to perform 16 index calculations and log subtractions in a data-parallel fashion.

Table 1: AVX-512 SIMD instructions used.

AVX-512 intrinsic	Pseudo-function
`_mm512_load_epi32 (void const* mem_addr)`	simd_load
`_mm512_store_epi32 (void* mem_addr, __m512i a)`	simd_store
`_mm512_add_epi32 (__m512i a, __m512i b)`	simd_add
`_mm512_sub_epi32 (__m512i a, __m512i b)`	simd_sub
`_mm512_reduce_min_epi32 (__m512i a)`	simd_minimum
`_mm512_i32gather_epi32 (__m512i vindex, void const* base_addr, int scale)`	simd_gather
`_mm512_i32scatter_epi32 (void* base_addr, __m512i vindex, __m512i a, int scale)`	simd_scatter

3.1 SIMD-based Block Sieving

In Intel's AVX-512 intrinsics, a special data type `__m512i` can store sixteen 32-bit integers in a vector. In (Sengupta and Das, 2017), vectorization of the subtraction phase of the sieve array is avoided in line and lattice sieving because of limitations of AVX. With AVX-512, we can perform SIMD-level parallelization of both index calculations and sieve-array modifications. We pack 16 primes $p_i, p_{i+1}, \ldots, p_{i+15} = p[i : i+15]$ from the factor base FB into a `__m512i` SIMD variable Δ_p, and their log values into another SIMD variable $\Delta_{\log p}$. For a fixed b, we calculate the starting sieving locations $a_{s_i}, a_{s_{i+1}}, \ldots, a_{s_{i+15}} = a_s[i : i+15]$ for $(r[i : i+15], p[i : i+15])$. Then, we pack another `__m512i` variable Δ_a with $a_s[i : i+15]$. Now, we keep on incrementing Δ_a by Δ_p over the entire a-line up to MAX_A to find out 16 sieving cell indices at a time. Using the AVX-512 intrinsic *gather*, we collect the values $S[\Delta_a]$ and store them in Δ_S, and subtract $\Delta_{\log p}$ from Δ_S. Then, we store the subtracted components back to their corresponding locations using another AVX-512 intrinsic *scatter*. This enables us to shorten the outer factor-base loop by a factor of 16 at the cost of some SIMD overhead. The index vector Δ_a is packed with the starting indices only once for a particular $p[i : i+15]$.

Moreover, the incremental addition of Δ_p to Δ_a does not require unpacking of any of the SIMD registers. Therefore we achieve effective vectorization of sieving-index calculations with 16-fold speedup. However, gathering and scattering costs after each index increment introduce some overhead. Algorithm 1 elaborates the steps of AVX-512-based block sieving. Table 1 lists the AVX-512 intrinsics used in the implementation of this algorithm.

Table 2: Details of the pseudo functions.

Pseudo-function	Description
allocate_memory	Allocates memory to the array.
is_incomplete	Checks if an element has pending iterations.
process_bucket	For each $(a, \log p)$ stored in the bucket, $S[a + MAX_A] - \log p$ is performed emptying the bucket.
insert_element	Inserts $(a, \log(p))$ into a given bucket.
populate_sieve_array	For a given b and $a \in [A_L, A_R]$, $S[a + MAX_A]$ is populated by $\log\lvert(-b)^d f(-a/b)\rvert$.
initialize	Initializes array elements with given value.

Algorithm 1: SIMD-based block sieving.

```
1  for b ← B_L to B_R do
2      for A_L ← −MAX_A to MAX_A in steps of b_s do
3          A_R ← minimum(A_L + b_s, MAX_A)
4          populate_sieve_array (S, b, A_L, A_R)
5          for each p[i : i+15] ∈ FB such that p[j] ≤ FB_S,
               i ≤ j ≤ i+15, do
6              Δ_p ← simd_load(p[i : i+15])
7              Δ_logp ← simd_load(log_p[i : i+15])
8              if A_L equals −MAX_A then
9                  a_s[i : i+15] ← initial sieving points
10             Δ_a ← simd_load(a_s[i : i+15])
11             while simd_minimum(Δ_a) ≤ A_R do
12                 Δ_S ← simd_gather(S, Δ_a)
13                 Δ_S ← simd_sub(Δ_S, Δ_logp)
14                 simd_scatter(S, Δ_S, Δ_a)
15                 Δ_a ← simd_add(Δ_a, Δ_p)
16             a_s[i : i+15] ← simd_store(Δ_a)
17             for j ← i to i+15 do
18                 if is_incomplete(a_s[j], A_R) then
19                     while a_s[j] ≤ A_R do
20                         S[a_s[j] + MAX_A] ←
                               S[a_s[j] + MAX_A] − log(p[j])
21                         a_s[j] ← a_s[j] + p[j]
```

3.2 SIMD-based Bucket Sieving

In bucket sieving, updating indices are calculated separately and stored in buckets. Later, the buckets are emptied followed by sieve-array updates. The bucket-filling part is SIMD-friendly. In bucket sieving, we work with the large primes ($p > FB_S$) of the factor base FB. We take 16 primes $p[i : i+15]$ at a time, and store them in an SIMD variable Δ_p. We also calculate the initial locations $a_s[i : i+15]$, and store them in another SIMD variable Δ_a. Then, we keep on finding 16 new sieving locations using an SIMD increment of Δ_a by Δ_p, and fill the buckets.

We start by allocating memory to the array *BARR* of buckets. In order to keep track of the numbers of elements in the buckets in the array *BARR*, we maintain another array B_T. For efficient memory usage, we pre-allocate each bucket in the bucket array *BARR* with a maximum element capacity of BUC_{MAX}. The

value of BUC_{MAX} is determined according to the size of the cache memory so that during the bucket-pop operation, cache-miss rates are minimized. During each insertion, we keep a check whether any bucket exceeds its capacity. If it so happens, we pop all the elements from that bucket, and update the sieving array at the stored locations. This strategy also eliminates the need of `malloc` and `free` operations of bucket entries after individual insert and pop operations. These memory operations are atomic, so avoiding them inside the loop boosts parallelism in multi-threaded implementations.

Algorithm 2 summarizes these implementation ideas. The workings of the pseudo-functions used in this algorithm are explained in Table 2.

4 EXPERIMENTAL RESULTS

4.1 Hardware and Software Setup

We use Intel's Xeon Gold Series (Model No. 6130) processor clocked at 2.10 GHz with an L3 Cache of size 22 MB. The gcc compiler (version 9.2.0), GMP library (version 6.1.2)and OpenMP API (version 4.5) is used. For calculating the prime ideals, we use Victor Shoup's NTL library (version 11.3.2) (Shoup et al., 2020). The optimization flag `-O3` and the intrinsic flag `-mavx=native` are used. In the multi-core implementations, we use all of the 16 cores of a single processor. The operating system is CentOS Linux release 7.4.1708 (Core).

4.2 Data Setup

As a test bench, we here consider the two numbers RSA-512 and RSA-768 which are factored as reported in (Cavallar et al., 2000) and (Kleinjung et al., 2010). In each of the cases, we consider the same polynomials that are used in the actual factorization attempts. Suitable partitioning of the factor base between block- and bucket-sieving primes has a major impact on the overall running time. We vary the small-versus-large demarcation boundary FB_S based on the sieving range MAX_A across various test cases.

For our multi-threaded implementation, we use the OpenMP directive `#pragma omp parallel for` to launch 16 threads expected to map to the individual cores. We allocate different segments of b values to the different threads in order to avoid concurrent writes. The read-only p and $\log p$ arrays are shared by all the threads, so that they can stay loaded in the cache. We have chosen the same limiting values (upper) for both the factor bases: $B_r = B_a = MAX_{FB}$.

4.3 Timing Results

Table 3 reports the timings $T_{\pm b}^{\pm v}$ of our implementations of sieving. The subscript indicates whether cache-friendly (block/bucket) sieving is used ($+b$) or not ($-b$), whereas the superscript indicates whether vectorization is used ($+v$) or not ($-v$). For example, T_{+b}^{-v} indicates the timing of our non-vectorized implementation with block and bucket sieving. All the times are in seconds, and stand for the combined times of rational sieving and algebraic sieving. Each sieving includes the time taken by the precomputation of initial indices, index increments and log subtractions, and locating potential sieving locations. The time for final trial divisions (relation generation) is excluded here. The number of threads utilized is denoted as N_θ. Each of the reported times is the average over 100 test cases.

Based on these four sets of timings, we calculate four relevant sets of speedup figures. The speedup of $[T+]$ over $[T-]$ is calculated as $\left(\frac{[T-]-[T+]}{[T-]}\right) \times 100\%$, where both the signs $\pm$ appear either in the subscript or in the superscript with the other kept unchanged. For example, $\psi_{+b} = \left(\frac{T_{+b}^{-v} - T_{+b}^{+v}}{T_{+b}^{-v}}\right) \times 100\%$ indicates the speedup obtained by vectorization on cache-friendly sieving, and $\psi^{-v} = \left(\frac{T_{-b}^{-v} - T_{+b}^{-v}}{T_{-b}^{-v}}\right) \times 100\%$ indicates the speedup obtained by cache-friendly sieving without vectorization.

The experimental data establishes two facts. First, AVX-512-based vectorization achieves a speedup of up to 56% in non-cache-friendly sieving and up to 25% in cache-friendly block and bucket sieving over non-vectorized implementations. Second, the effectiveness of cache-friendly sieving is manifested by a speedup of up to 63% both with and without vectorization. In particular, the best running times are obtained with both cache-friendly sieving and vectorization (the column headed T_{+b}^{+v}).

5 CONCLUSION

In this paper, we report the practical effectiveness of block and bucket sieving and AVX-512-based vectorization. This study establishes the usefulness of exploiting latest hardware features for implementing time-consuming algorithms like the GNFSM for factoring integers. There are several ways in which our study can be extended. Both cache-friendly sieving

Algorithm 2: SIMD-based bucket sieving.

```
1  B_N ← (2×MAX_A + 1)/B_s                                   // Total number of buckets
2  BARR ← allocate_memory(B_N × BUC_MAX)                     // Memory for all buckets
3  B_T ← initialize(0)                                       // All buckets are initially empty
4  for b ← B_L to B_R do
5      for A_L ← −MAX_A to MAX_A in steps of b_s do
6          A_R ← minimum(A_L + b_s, MAX_A)
7          populate_sieve_array (S, b, A_L, A_R)             // Initialize with log values
8          block_sieving(A_L, A_R)                           // Use Algorithm 1 to handle small primes
9      for each p[i : i+15] ∈ FB_LIST such that p[j] > FB_S, i ≤ j ≤ i+15 do
10         Δ_p ← simd_load(p[i : i+15])
11         a_s[i : i+15] ← initial sieving points
12         Δ_a ← simd_load(a_s[i : i+15])                    // Calculate next indices
13         while simd_minimum(Δ_a) ≤ MAX_A do
14             for each (a_s[j], p[j]) in (a_s[i : i+15], p[i : i+15]) do
15                 b_n ← (a_s[j] + MAX_A)/B_s                // The bucket number
16                 if B_T[b_n] equals BUC_MAX then
17                                                           // Bucket capacity reached
18                     process_bucket(S, BARR, B_T, b_n)
19                     B_T[b_n] ← 0                          // Bucket flushed
20                 insert_element(S, BARR, B_T, b_n, a_s[j], log(p[j]))
21                 B_T[b_n] ← B_T[b_n] + 1                   // Update entry stored
22             Δ_a ← simd_add(Δ_a, Δ_p)
23             a_s[i : i+15] ← simd_store(Δ_a)
24         for j ← i to i+15 do
25             if is_incomplete(a_s[j], MAX_A) then
26                 while a_s[j] < MAX_A do
27                     b_n ← (a_s[j] + MAX_A)/B_s
28                     if B_T[b_n] equals BUC_MAX then
29                         process_bucket(S, BARR, B_T, b_n)
30                         B_T[b_n] ← 0
31                     insert_element(S, BARR, B_T, b_n, a_s[j], log(p[j]))
32                     B_T[b_n] ← B_T[b_n] + 1
33                     a_s[j] ← a_s[j] + p[j]
34         for b_n ← 1 to B_N do
35             process_bucket(S, BARR, B_T, b_n)             // Use all non-empty buckets
```

and the use of vectorization are expected to boost lattice sieving by the same margins as line sieving. However, explicit experiments are not carried out with lattice sieving. Block sieving is effectively vectorized, but bucket sieving has further rooms for investigation, particularly in the bucket emptying process.

The current processor technology imposes restrictions on the processing speed in the presence of SIMD utilization. For Xeon 6130 processors, individual cores work at a speed of 2.1 GHz, but enabling AVX2 or AVX-512 reduces the frequency to 60–65% (WikiChip, 2020). Further reduction happens with the increasing number of cores. This is one of the main reasons behind not achieving the ideal speedup in the case of multi-threaded implementations. Finding a balance between the use of multiple cores and the use of SIMD features remains a challenging practical area of investigation.

REFERENCES

Aoki, K. and Ueda, H. (2004). Sieving using bucket sort. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 92–102. Springer.

Table 3: Timing and speedup figures.

n	Parameters					Sieving time (sec)				Percentage speedup			
	MAX_A	MAX_B	N_θ	FB_S	MAX_{FB}	T_{-b}^{-v}	T_{+b}^{-v}	T_{-b}^{+v}	T_{+b}^{+v}	ψ_{-b}	ψ_{+b}	ψ^{-v}	ψ^{+v}
RSA-512	5×10^5	11	1	4.5×10^4	5×10^4	0.281	0.131	0.151	0.098	46.26	25.19	53.38	35.10
	3×10^6	11	1	4.5×10^4	5×10^4	2.056	0.968	1.270	0.737	38.23	23.86	52.92	41.97
	3×10^6	11	1	2^{18}	3×10^5	2.255	1.068	1.300	0.805	42.35	24.63	52.64	38.08
	3×10^6	161	16	2^{18}	3×10^5	6.299	2.321	5.929	2.212	5.87	4.70	63.15	62.69
RSA-768	5×10^5	11	1	4.5×10^4	5×10^5	0.386	0.192	0.171	0.160	55.70	16.67	50.26	6.43
	3×10^6	11	1	4.5×10^4	5×10^5	2.506	1.368	1.332	1.137	46.85	16.89	45.41	14.64
	3×10^6	11	1	2^{19}	3×10^6	2.947	1.427	1.511	1.186	48.73	16.89	51.58	21.51
	3×10^6	161	16	2^{19}	3×10^6	7.499	4.072	6.691	3.763	10.77	7.59	45.70	43.76

Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., and Zimmermann, P. (2020a). Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. *arXiv preprint arXiv:2006.06197*.

Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., and Zimmermann, P. (2020b). Factorization of rsa-250. https://caramba.loria.fr/rsa250.txt. Accessed: 2021-02-08.

Briggs, M. E. (1998). *An introduction to the general number field sieve*. PhD thesis, Virginia Tech.

Buhler, J. P., Lenstra, H. W., and Pomerance, C. (1993). Factoring integers with the number field sieve. In *The development of the number field sieve*, pages 50–94. Springer.

Cavallar, S., Dodson, B., Lenstra, A. K., Lioen, W., Montgomery, P. L., Murphy, B., Te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., et al. (2000). Factorization of a 512-bit rsa modulus. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–18. Springer.

Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., et al. (2010). Factorization of a 768-bit rsa modulus. In *Annual Cryptology Conference*, pages 333–350. Springer.

Lenstra, A. K., Hendrik Jr, W., et al. (1993a). *The development of the number field sieve*, volume 1554. Springer Science & Business Media.

Lenstra, A. K., Lenstra, H. W., Manasse, M. S., and Pollard, J. M. (1993b). The factorization of the ninth fermat number. *Mathematics of Computation*, 61(203):319–349.

Lenstra, A. K., Lenstra Jr, H. W., Manasse, M. S., and Pollard, J. M. (1990). The number field sieve. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 564–572. ACM.

Pollard, J. M. (1993). The lattice sieve. In *The development of the number field sieve*, pages 43–49. Springer.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Sengupta, B. and Das, A. (2017). Use of simd-based data parallelism to speed up sieving in integer-factoring algorithms. *Applied Mathematics and Computation*, 293:204–217.

Shoup, V. et al. (2020). Ntl: A library for doing number theory.

Wambach, G. and Wettig, H. (1995). *Block sieving algorithms*. Citeseer.

WikiChip (2020). Intel xeon gold 6130. https://en.wikichip.org/wiki/intel/xeon_gold/6130. Accessed: 2021-02-08.