Use of SIMD-Based Data Parallelism to Speed up Sieving in Integer-Factoring Algorithms [⋆]

Binanda Sengupta and Abhijit Das

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, West Bengal, PIN: 721302, India
`binanda.sengupta,abhij@cse.iitkgp.ernet.in`

Abstract. Many cryptographic protocols derive their security from the apparent computational intractability of the integer factorization problem. Currently, the best known integer-factoring algorithms run in subexponential time. Efficient parallel implementations of these algorithms constitute an important area of practical research. Most reported implementations use multi-core and/or distributed parallelization. In this paper, we use SIMD-based parallelization to speed up the sieving stage of integer-factoring algorithms. We experiment on the two fastest variants of factoring algorithms: the number-field sieve method and the multiple-polynomial quadratic sieve method. Using Intel's SSE2 and AVX intrinsics, we have been able to speed up index calculations in each core during sieving. This performance enhancement is attributed to a reduction in the packing and unpacking overheads associated with SIMD registers. We handle both line sieving and lattice sieving. We also propose improvements to make our implementations cache-friendly. We obtain speedup figures in the range 5–40%. To the best of our knowledge, no public discussions on SIMD parallelization in the context of integer-factoring algorithms are available in the literature.

Keywords: Integer Factorization, Sieving, Multiple-Polynomial Quadratic Sieve Method, Number-Field Sieve Method, Single Instruction Multiple Data, Streaming SIMD Extensions, Advanced Vector Extensions

[⋆] Part of this work is presented at SPACE 2013.

1 Introduction

Let n be a large composite integer having the factorization

$$n = p_1^{v_{p_1}} p_2^{v_{p_2}} \cdots p_k^{v_{p_k}} = \prod_{i=1}^{k} p_i^{v_{p_i}}.$$

The integer factorization problem deals with the determination of all the prime divisors $p_1, p_2, \ldots, p_k$ of n and their corresponding multiplicities $v_{p_1}, v_{p_2}, \ldots, v_{p_k}$. The earlier algorithms proposed to solve this problem run in exponential time in the input size (the number of bits in n, that is, $\log_2 n$ or $\lg n$). Modern factoring algorithms have subexponential running times of the form

$$L(n, \omega, c) = \exp\left((c + \mathrm{o}(1))(\ln n)^{\omega}(\ln \ln n)^{1-\omega}\right),$$

where c and ω are positive real constants with $0 < \omega < 1$. The smaller the constants ω and c are, the faster these algorithms are. The fastest known general-purpose factoring algorithm is the (general) number-field sieve method (NFSM) for which $\omega = 1/3$ and $c = (64/9)^{1/3} \approx 1.923$. The multiple-polynomial quadratic sieve method (MPQSM) is reported as the second fastest factoring algorithm. Its running time corresponds to $\omega = 1/2$. Although the NFSM is theoretically faster than the MPQSM, the asymptotic behavior shows up only when the input integer n is moderately large (having at least a few hundred bits). Better than fully exponential-time algorithms as they are, these subexponential algorithms are still prohibitively slow, and it is infeasible to factor a 1024-bit composite integer in a reasonable amount of time (like a few years). Many cryptographic protocols (including RSA) derive their security from this apparent intractability of the integer-factoring problem.

The most time-consuming part in the NFSM and the MPQSM is the sieving stage in which a large number of candidates (suitably small positive or negative integers) are generated. Only those candidates which factor completely over a predetermined set of small primes yield useful relations. We combine these relations using linear-algebra tools in order to arrive at a congruence of the form $x^2 \equiv y^2 \pmod{n}$. If $x \not\equiv \pm y \pmod{n}$, the nontrivial factor $\gcd(x - y, n)$ is discovered.

In this paper, we deal with efficient implementations of the sieving stage in the NFSM and the MPQSM. Sieving turns out to be massively parallelizable on multi-core and even distributed platforms. All recent implementations of sieving in the literature deal with issues of such large-scale parallelization. In this paper, we look at the problem from a different angle. We plan to exploit SIMD features commonly available in modern desktop and server machines. We investigate whether SIMD-based parallelization can lead to some speedup in the computation of each core. The sieving procedure involves two data-parallel operations: computation of indices and subtraction of log values. We theoretically and experimentally demonstrate that the index-calculation process can be effectively parallelized by SIMD intrinsics. This effectiveness banks on that we can avoid packing of data in SIMD registers in each iteration of the sieving loop. The subtraction operations, on the other hand, do not benefit from such parallelization efforts. First, we need packing and unpacking of SIMD registers in each iteration of the loop.

Second, the individual data items that are packed in SIMD registers do not belong to contiguous locations in the memory.

The main technical contribution of this paper is the successful parallelization of index calculations in the sieving stage using Intel's SSE2 (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) features on a Sandy Bridge platform. We have been able to speed up the sieving stage of both the NFSM and the MPQSM by 5–40%. Intel's recently released AVX2 instruction set is expected to increase this speedup further (AVX uses 256-bit vectorization only on floating-point values, whereas AVX2 has this feature for integer operands too). We study two variants of sieving—the line sieve and the lattice sieve—in the NFSM implementation. We also propose a cache-friendly implementation strategy. To the best of our knowledge, no SIMD-based parallelization attempts on sieving algorithms for integer factorization are reported in the literature. The implementations described by [1] and [2] use the term SIMD but are akin to multicore parallelization in a 16K MasPar SIMD machine with a 128×128 array of processing elements. In this paper, we show that the use of SSE2 and AVX instruction sets available in modern processors can achieve an additional speedup on each these cores.

The rest of the paper is organized as follows. Section 2 introduces the background relevant for the remaining sections, namely, the sieving procedures in the MPQSM and in the NFSM, the lattice sieve, and Intel's SSE2 and AVX components. In Section 3, we detail our implementation strategies for the MPQSM and the NFSM sieves. Section 4 presents our experimental results achieved on a Sandy Bridge platform. We analyze the speedup figures, and demonstrate the effectiveness of our cache-friendly implementation. We conclude the paper in Section 5 after highlighting some possible future extensions of our work.

2 Background

2.1 A Summary of Known Integer-Factoring Algorithms

Most modern integer-factoring algorithms construct Fermat congruences of the form $x^2 \equiv y^2 \pmod{n}$. If $x \not\equiv \pm y \pmod{n}$, then $\gcd(x-y,n)$ is a non-trivial factor of n. If these congruences are available for many random values of x and y, at least some of these pairs is/are expected to reveal non-trivial factors of n with high probability. In order to generate such congruences, one collects many relations, each of which supplies a linear equation modulo 2. Relation collection is efficiently carried out by sieving. After sufficiently many relations are collected, the linear equations are combined to find non-zero vectors in the nullspace of the system. Each such non-zero vector yields a Fermat congruence. The factoring algorithms differ from one another in the way the relations are generated, that is, in their sieving techniques. The linear-algebra phase turns out to be identical in all these algorithms.

J. D. Dixon [3] proposes the simplest variant of such a factoring method. Based on the work of Lehmer and Powers [4], Morrison and Brillhart introduce another variant known as the CFRAC method [5], where relations are obtained from the continued fraction expansion of $\sqrt{n}$.

In Pomerance's quadratic sieve method (QSM) [6], the polynomial $T(c) = J + 2Hc + c^2$ (where $H = \lceil \sqrt{n} \rceil$ and $J = H^2 - n$) is evaluated for small values of c (in the range $-M \leqslant c \leqslant M$). If some $T(c)$ value factors completely over the first t primes $p_1, p_2, \ldots, p_t$, we get a relation. In Dixon's method, the smoothness candidates are $\mathrm{O}(n)$, whereas in CFRAC and QSM, these are $\mathrm{O}(\sqrt{n})$, resulting in a larger proportion of smooth integers (than Dixon's method) in the pool of smoothness candidates. Moreover, QSM replaces trial divisions by sieving (subtractions after some preprocessing). This gives QSM a better running time than Dixon's method and CFRAC.

R. D. Silverman introduces a variant of quadratic sieve method, called the multiple polynomial quadratic sieve method (MPQSM) [7]. Instead of using the fixed polynomial $T(c)$, the MPQSM uses a more general polynomial $T(c) = Wc^2 + 2Vc + U$ so that the smoothness candidates are somewhat smaller than those in the QSM.

The number field sieve method (NFSM) is originally proposed for integers of a special form [8], and is later extended to factor arbitrary integers [9]. Pollard introduces the concept of lattice sieving [10] as an efficient implementation of the sieves in the NFSM. The conventional sieving is called line sieving.

Some other methods for factoring large integers include the cubic sieve method (CSM) [11] and the elliptic curve method (ECM) [12].

The linear-algebra phase in factoring algorithms can be reasonably efficiently solved using sparse system solvers like the block Lanczos method [13]. We do not deal with this phase in this paper. We focus our attention only to the relation-collection stage (more precisely, the sieving part of it).

2.2 A Brief Introduction to SIMD Features

All our implementations of sieving use SIMD (Single Instruction Multiple Data) parallelization. We carry out our experiments on a Sandy Bridge architecture from Intel. This architecture provides two types of vectorization primitives SSE2 and AVX.

SSE (Streaming SIMD Extensions) [14] is an extension of the previous x86 instruction set, and SSE2 enhances the SSE instruction set further. SSE2 instructions apply to 128-bit SIMD registers (XMM). In each XMM register, we can accommodate multiple data of some basic types (like four 32-bit integers, four single-precision floating-point numbers, and two double-precision floating-point numbers). There are special instructions to perform vector operations (like arithmetic, logical, shift, conversion, and comparison) on these XMM registers. The basic idea to exploit this architecture is to pack these registers with multiple data, perform a single vector instruction, and finally unpack the output XMM register to obtain the desired individual results.

AVX (Advanced Vector Extensions) [15] is a recent extension to the general x86 instruction set. It is introduced in Intel's Sandy Bridge processor. This architecture is designed with sixteen 256-bit SIMD registers (YMM). Now, we can accommodate eight single-precision or four double-precision floating-point numbers in one YMM register. AVX is designed with three-operand non-destructive SIMD instructions, where the destination register is different from the two source registers.

Another important point to note here is that mixing of AVX instructions and legacy (that is, non-VEX-encoded) SSE instructions is not recommended. XMM registers are aliased as the lower 128 bits of YMM registers. Every AVX-SSE or SSE-AVX transition incurs severe clock-cycle penalties. To eliminate AVX-SSE transitions, legacy SSE instructions can be converted to their equivalent VEX-encoded instructions by zeroing the upper 128 bits of the YMM registers, as suggested in [16].

The AVX instruction set is currently applicable for only floating-point operations. Intel's Haswell architecture incorporates another extension AVX2 which supports instructions for integer operations on 256-bit registers. Programming languages come with intrinsics for high-level access to SIMD instructions. We can use these intrinsics directly in our implementations to exploit data parallelism.

2.3 The MPQSM Sieve

The MPQSM is a variant of the quadratic sieve method (QSM). Instead of using a single polynomial (with fixed coefficients), the MPQSM deals with a general polynomial and tunes its coefficients to generate small smoothness candidates. This variant is parallelizable in the sense that different polynomials can be assigned to different cores.

The MPQSM sieve deals with a polynomial

$$T(c) = Wc^2 + 2Vc + U \tag{1}$$

with $V^2 - UW = n$. The factor base consists of the first t primes $p_1, p_2, \ldots, p_t$, where t is chosen based on a bound B. Only those primes are included in the factor base, modulo which n is a quadratic residue. First, we calculate the values of $T(c)$ for all c in the range $-M \leqslant c \leqslant M$. Now, we try to find those values of c, for which $T(c)$ is B-smooth, that is, $T(c)$ factors completely into primes $\leqslant B$. If $T(c) = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t}$ for some non-negative integral values of α_i, then by multiplying Eqn (1) by W we get the relation

$$(Wc+V)^2 \equiv W p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t} \pmod{n}. \tag{2}$$

We include W too in the factor base.

Now, we briefly discuss the sieving step in the MPQSM. The smoothness candidates are the values $T(c)$ of Eqn (1) for all c in the range $[-M, M]$. We need to locate those values of c for which $T(c)$ is smooth over the factor base. We take an array A indexed by c. Initially, we store $\log|T(c)|$ in $A[c]$, truncated after three decimal places. Indeed, we can avoid floating-point operations by storing $\lfloor 1000 \log|T(c)| \rfloor$.

After this initialization, we try to find solutions of the congruence $T(c) \equiv 0 \pmod{p^h}$, where p is a small prime in the factor base, and h is a small positive exponent. The solutions of the congruence

$$Wc^2 + 2Vc + U \equiv 0 \pmod{p^h},$$

are

$$c \equiv \frac{-2V \pm \sqrt{4V^2 - 4UW}}{2W} \equiv \frac{-V \pm \sqrt{n}}{W} \equiv W^{-1}(-V \pm \sqrt{n}) \pmod{p^h}. \tag{3}$$

For $h = 1$, we use a root-finding algorithm to compute the square roots of n modulo p (see [17, 18]). For $h > 1$, we obtain the solutions modulo p^h by lifting the solutions modulo p^{h-1} (see [19]).

Let s_1, s_2 be the solutions of $T(c) \equiv 0 \pmod{p^h}$ chosen to be as small as possible in the range $[-M, M]$. Then, all the solutions of $T(c) \equiv 0 \pmod{p^h}$ are $s_1 + jp^h$ and

$s_2 + jp^h$ for $j = 0, 1, 2, 3, \ldots$. We subtract $\lfloor 1000 \log p \rfloor$ from these array locations. For $p = 2$ and $h = 1$, there is only one initial solution of the congruence. For $p = 2$ and $h > 1$, there may be more than two initial solutions.

When all appropriate values of p^h are considered, the array locations storing $A[c] \approx 0$ correspond to the smooth values of $T(c)$. We apply trial division on these smooth values of $T(c)$. If some $T(c)$ value is not smooth, then the corresponding array entry $A[c]$ holds an integer $\geqslant \lfloor 1000 \log p_{t+1} \rfloor$ for the smallest prime $p_{t+1} > B$. This justifies the correctness of the calculations using approximate (truncated) log values.

2.4 The NFSM Sieve

The NFSM [9] involves a monic irreducible polynomial $f(x) \in \mathbb{Z}[x]$ of a small degree d and an integer $m \approx n^{1/d}$ such that $f(m) \equiv 0 \pmod{n}$, n being the integer to be factored. One possibility is to take $m = \lfloor n^{1/d} \rfloor$, and express n in base m as $n = m^d + c_{d-1}m^{d-1} + \cdots + c_0$, with the integers c_i varying in the range 0 to $m-1$. For this choice, we take $f(x) = x^d + c_{d-1}x^{d-1} + \cdots + c_0$, provided that it is an irreducible polynomial in $\mathbb{Z}[x]$. We have $f(m) = n$, implying that $f(m) \equiv 0 \pmod{n}$, as desired.

If $\theta \in \mathbb{C}$ is a root of a monic irreducible polynomial $f(x)$ of degree d with integral coefficients and $m \in \mathbb{Z}$ is an integer such that $f(m) \equiv 0 \pmod{n}$, then there exists a ring homomorphism $\phi : \mathbb{Z}[\theta] \to \mathbb{Z}_n$ defined by $\phi(\theta) = m \pmod{n}$ (and $\phi(1) = 1 \pmod{n}$).[1]

Now, let E be a set of pairs of integers (a, b) satisfying

$$\prod_{(a,b)\in E} (a + b\theta) = \alpha^2,$$
$$\prod_{(a,b)\in E} (a + bm) = y^2,$$

for some $\alpha \in \mathbb{Z}[\theta]$ and $y \in \mathbb{Z}$. Let $\phi(\alpha) = x \in \mathbb{Z}_n$. Then, we get the Fermat congruence

$$\begin{aligned} x^2 &\equiv \phi(\alpha)^2 \equiv \phi(\alpha^2) \equiv \phi\Big(\prod_{(a,b)\in E} (a + b\theta)\Big) \\ &\equiv \prod_{(a,b)\in E} \phi(a + b\theta) \equiv \prod_{(a,b)\in E} (a + bm) \equiv y^2 \pmod{n}. \end{aligned}$$

The NFSM uses a rational factor base (RFB) and an algebraic factor base (AFB). The RFB consists of the first t_1 primes $p_1, p_2, \ldots, p_{t_1}$, where t_1 is chosen based on a bound B_{rat}. The AFB consists of some primes of small norms in $\mathfrak{O}$. Application of the homomorphism ϕ lets us rewrite the AFB in terms of t_2 rational (integer) primes $p_1, p_2, \ldots, p_{t_2}$, where t_2 is chosen based on a bound B_{alg}.

Now, we describe the rational sieve and the algebraic sieve of the NFSM. Here, we deal with incomplete sieving (see [19]), that is, higher powers of factor-base primes are not considered in sieving.

[1] $\mathbb{C}$ denotes the field of complex numbers, and $\mathbb{Q}$ the field of rational numbers. For an algebraic element $\theta \in \mathbb{C}$, $\mathbb{Q}(\theta)$ stands for the field obtained by adjoining θ to $\mathbb{Q}$. $\mathfrak{O}$ is the set of all algebraic integers in $\mathbb{Q}(\theta)$. $\mathbb{Z}[\theta]$, a subring of $\mathfrak{O}$, is the set of all $\mathbb{Z}$-linear combinations of the elements $1, \theta, \theta^2, \ldots, \theta^{d-1}$.

Let $T(a,b) = a + bm$. First, we calculate the values of $T(a,b)$ for all b in the range $1 \leqslant b \leqslant u$ and a in the range $-u \leqslant a \leqslant u$. Now, we try to find those (a,b) pairs with $\gcd(a,b) = 1$ and $b \not\equiv 0 \pmod{p}$, for which $T(a,b)$ is B_{rat}-smooth, that is, $T(a,b)$ factors completely into the primes $p_1, p_2, \ldots, p_{t_1}$. The determination whether a small prime p_i divides some $a + bm$ is equivalent to solving the congruence $a + bm \equiv 0 \pmod{p_i}$. The sieving bound u is determined based upon certain formulas which probabilistically guarantee that we can obtain the requisite number of relations from the entire sieving process.

We take a two-dimensional array A indexed by a and b. Initially, we store $\log|T(a,b)|$ in $A[a,b]$, truncated after three decimal places (or the integers $\lfloor 1000 \log|T(a,b)| \rfloor$). After this initialization, we find solutions of the congruence $T(a,b) \equiv 0 \pmod{p}$, where p is a small prime in the RFB. For a fixed b, the solutions are $a \equiv -bm \pmod{p}$. Let s be the least possible solution of $T(a,b) \equiv 0 \pmod{p}$ in the range $[-u,u]$ for a particular b. Then, all the solutions of $T(a,b) \equiv 0 \pmod{p}$ for that b are $s + jp$, $j = 0,1,2,3,\ldots$. We subtract $\lfloor 1000 \log p \rfloor$ from all the array locations $A[a,b]$ such that $a = s + jp$. We repeat this procedure for all small primes p in the RFB and for all allowed values of b. As we deal with only small primes with exponent $h = 1$, the array locations storing $A[a,b]$ values less than $\lfloor 1000 \xi \log p_{t_1+1} \rfloor$ (where p_{t_1+1} is the smallest prime greater than B_{rat}, and ξ lies between 1.0 and 2.5) are subjected to trial division to verify the smoothness of $T(a,b)$.

The algebraic sieve uses the norm function $\mathrm{N} : \mathbb{Q}(\theta) \to \mathbb{Q}$. Its restriction to $\mathbb{Z}[\theta]$ yields norm values in $\mathbb{Z}$. For an element of the form $a + b\theta \in \mathbb{Z}[\theta]$, we have the explicit formula:

$$\mathrm{N}(a + b\theta) = (-b)^d f(-a/b) = a^d - c_{d-1} a^{d-1} b + \cdots + (-1)^d c_0 b^d,$$

where $f(x) = x^d + c_{d-1} x^{d-1} + \cdots + c_0$.

An element $\alpha \in \mathbb{Z}[\theta]$ is smooth with respect to the small primes of $\mathfrak{O}$ if and only if $\mathrm{N}(\alpha) \in \mathbb{Z}$ is B_{alg}-smooth. For each small prime p in the AFB, we compute the set of zeros of f modulo p, that is, all r values satisfying the congruence $f(r) \equiv 0 \pmod{p}$. For a particular b with $b \not\equiv 0 \pmod{p}$ and $1 \leqslant b \leqslant u$, the norm values with $\mathrm{N}(a + b\theta) \equiv 0 \pmod{p}$ correspond to the a values given by $a \equiv -br \pmod{p}$ for some root r of f modulo p. It follows that the same sieving technique as discussed for the rational sieve can be easily adapted to the case of the algebraic sieve.

An (a,b) pair for which both $a + bm$ and $a + b\theta$ are smooth gives us a relation. Sufficiently many relations obtained from the two sieves are combined using linear algebra to get the set E of (a,b) pairs such that $\prod_{(a,b) \in E} (a + bm) = y^2$ and $\prod_{(a,b) \in E} (a + b\theta) = \alpha^2$.

2.5 The Lattice Sieve Method

Pollard [10] introduces the lattice sieve method based on an idea proposed in [20]. Here, we describe the lattice sieve (its sieving-by-rows variant) for the rational sieve only. The same technique can be used in the algebraic sieve too. First, we take the rational factor base B_{rat}, and partition B_{rat} in the following way. By S, we denote the set consisting of

Algorithm 1 : Lagrange's rank-2 lattice-basis reduction algorithm

```
Input : A basis $\{\mathbf{w}_1, \mathbf{w}_2\}$ of a two-dimensional lattice $L$
Output: A Lagrange-reduced basis $\{\mathbf{v}_1, \mathbf{v}_2\}$ of $L$
$\mathbf{v}_1 = \mathbf{w}_1$ and $\mathbf{v}_2 = \mathbf{w}_2$                                    ▷ Initialization
if ($||\mathbf{v}_1|| > ||\mathbf{v}_2||$) then              ▷ $||\mathbf{v}||$ denotes Euclidean norm of vector $\mathbf{v}$
    $swap(\mathbf{v}_1, \mathbf{v}_2)$
end if
while ($||\mathbf{v}_1|| < ||\mathbf{v}_2||$) do
    $u = (\mathbf{v}_1 \cdot \mathbf{v}_2)/(\mathbf{v}_1 \cdot \mathbf{v}_1)$            ▷ $u\mathbf{v}_1$ is the orthogonal projection of $\mathbf{v}_2$ on $\mathbf{v}_1$
                          ▷ Here, $\cdot$ denotes the inner or dot product of two vectors
    $\mathbf{v}_2 = \mathbf{v}_2 - round(u)\mathbf{v}_1$
    $swap(\mathbf{v}_1, \mathbf{v}_2)$
end while
$swap(\mathbf{v}_1, \mathbf{v}_2)$                                  ▷ To ensure $||\mathbf{v}_1|| \leqslant ||\mathbf{v}_2||$
Return $(\mathbf{v}_1, \mathbf{v}_2)$
```

all small primes p of the factor base with $p \leqslant B'_{rat}$. Moreover, we denote by M the set consisting of all medium primes q of the factor base with $B'_{rat} < q \leqslant B_{rat}$. Usually, B'_{rat} is selected in such a way that the ratio B'_{rat}/B_{rat} lies between 0.1 and 0.5.

Let $q \in M$ be a medium prime (also called a special q in the notation of [20]). Unlike the line sieve described in earlier chapters, we now take into account a region R in the (a,b) plane. For a particular q, we sieve only those (a,b) pairs which satisfy the congruence $a + bm \equiv 0 \pmod{q}$. Furthermore, we consider only the small primes p with $p < q$ while sieving the integers $a + bm$ for a fixed q.

All the pairs (a,b) satisfying $a+bm \equiv 0 \pmod{q}$ form a lattice L_q in the (a,b) plane. The vectors $\mathbf{w}_1 = (q,0)$ and $\mathbf{w}_2 = (-m,1)$ generate the lattice L_q. This basis is reduced using Lagrange's lattice-basis reduction algorithm for two-dimensional lattices (see [21, 22]). The algorithm is similar to Euclid's gcd computation and is given as Algorithm 1. The iterations of the **while** loop monotonically decrease the Euclidean norms of $\mathbf{v}_1$ and $\mathbf{v}_2$ by subtracting an appropriate multiple of the shorter vector from the other.

From Algorithm 1, we get a reduced basis $\{\mathbf{v}_1, \mathbf{v}_2\}$ of the lattice L_q with $\mathbf{v}_1$ being the shorter one. Let us write $\mathbf{v}_1 = (a_1, b_1)$ and $\mathbf{v}_2 = (a_2, b_2)$. All the points of L_q can be uniquely represented as $l_1\mathbf{v}_1 + l_2\mathbf{v}_2$ for $l_1, l_2 \in \mathbb{Z}$. The lattice sieving takes place in the (l_1, l_2) plane. Given a pair (l_1, l_2), we can retrieve the corresponding (a,b) pair as $(a,b) = (l_1a_1 + l_2a_2,\, l_1b_1 + l_2b_2)$.

We take two sieving bounds L_1 and L_2 and a two-dimensional array A indexed by l_1 and l_2. The indices vary in the ranges $-L_1 \leqslant l_1 \leqslant L_1$ and $1 \leqslant l_2 \leqslant L_2$. The array location (l_1, l_2) represents the value $a + bm = l_1u_1 + l_2u_2$, where $u_1 = a_1 + b_1m$ and $u_2 = a_2 + b_2m$ with $\gcd(u_1, u_2) = q$. As $\mathbf{v}_1$ is shorter than $\mathbf{v}_2$, we choose $L_1 > L_2$ in order to make sieving efficient. All the array elements are initialized to zero. Subsequently, $\lfloor 1000 \log p \rfloor$ values are added to $A[l_1, l_2]$ for all primes $p < q$ for which $l_1u_1 + l_2u_2 \equiv 0 \pmod{p}$. After all primes p are considered, the sum stored in $A[l_1, l_2]$ is subtracted from $\lfloor 1000 \log(a + bm) \rfloor$ for the corresponding (a,b) pair. If the difference is smaller than a threshold, trial division is used to verify whether $a + bm$ is actually smooth. The threshold is usually chosen to be $\lfloor 1000\xi \log p_{t_1+1} \rfloor$ for the smallest prime $p_{t_1+1} >$

B_{rat} and with $1.0 \leqslant \xi \leqslant 2.5$. This liberal selection is necessitated by the fact that the congruence $a + bm \equiv 0 \pmod{p^h}$ is now solved only for $h = 1$.

The above steps are repeated for each medium prime $q \in M$.

3 Implementation Details

Currently, the largest general-purpose integer that has been factored by a subexponential algorithm is a 768-bit (232-digit) RSA modulus (see [23]). Consequently, we restrict our experiments to integers with no more than 250 decimal digits. For the NFSM, we experiment with two integers n_1 and n_2 having 60 and 120 decimal digits, respectively. Each of these is like an RSA modulus (that is, a product of two primes of the same bit length). In the MPQSM, we have implemented the complete sieve, that is, the congruence $T(c) \equiv 0 \pmod{p^h}$ is solved for all relevant p, h values. For the NFSM, on the other hand, we have implemented only the incomplete sieve, that is, the value $h = 1$ is only considered. We have not gone for any sophisticated polynomial-selection procedure in the NFSM. We have instead chosen $f(x)$ to be the polynomial obtained from the m-ary representation of n, where $m = \lfloor n^{1/d} \rfloor$. As suggested in [2, 24], we take $d = 3$ if the number of digits in n is less than 80, and $d = 5$ for larger integers.

3.1 Sequential Implementations

The sequential implementations are straightforward. Let us consider the MPQSM sieve first. For a small prime p in the factor base, let H be the largest exponent h for which Eqn (3) is satisfied by some $c \in [-M, M]$. For each $h \in \{1, 2, \ldots, H\}$, these solutions are precomputed before the sieving loop. Let s be a solution for some p, h values. We take s to be the minimum location in the sieving interval $[-M, M]$. We sieve the array A by subtracting $\lfloor 1000 \log p \rfloor$ from all the array locations $c = s + jp^h$ with $j \in \mathbb{N}$. This procedure is repeated for all small primes in the factor base. Algorithm 2 presents a pseudocode of our sequential implementation of the MPQSM sieve.

In the NFSM, the rational and the algebraic sieves are carried out independently in two two-dimensional arrays. If for some pair (a, b) with $\gcd(a, b) = 1$, both $a + bm$ and $a + b\theta$ are found to be smooth from the two sieves, we obtain a relation. The number of relations depends on the sieving bound u. Our basic goal is to investigate the benefits of SIMD parallelization instead of generating a complete solvable system, so we have experimented with small sieving bounds only.

In the lattice sieve method, we choose a medium prime $q \in M$. Then, we determine a reduced basis $\{\mathbf{v}_1, \mathbf{v}_2\}$ for L_q using Algorithm 1 (see Section 2.5). Subsequently, the rational sieve is carried out in a two-dimensional array whose rows and columns are indexed by l_2 and l_1, respectively. We consider sieving by rows only, that is, we fix l_2, start with the least array location l_1 in the sieving interval satisfying $l_1 u_1 + l_2 u_2 \equiv 0 \pmod{p}$, and sieve all the other locations for the particular row l_2. We precompute $u_1^{-1} \pmod{p}$ to find the initial solutions for different l_2 values. If $u_1 \equiv 0 \pmod{p}$ or $u_2 \equiv 0 \pmod{p}$, we sieve all elements of every p-th row or every p-th column, respectively. We consider only the coprime (l_1, l_2) pairs. Finally, the gcd of every pair (a, b) indicating smoothness is calculated. If this gcd is one, the (a, b) pair provides a relation. If $\gcd(a, b) = q$,

Algorithm 2 Pseudocode for the sequential implementation of the MPQSM sieve

```
Input : Factor base $FB$, sieving bound $M$, and sieving array $A$ indexed by $c$
Output: Sieving array $A$ after subtractions of log values

for all $c$ in the range $-M \leqslant c \leqslant M$ do
    $A[c] \leftarrow \lfloor 1000 \log T(c) \rfloor$
end for
for all prime powers $p^h$ do                                ▷ Prime $p \in FB$ and $h$ small positive integer
    Let $s$ be a solution of of $T(c) \equiv 0 \pmod{p^h}$           ▷ These solutions are precomputed
    for all solutions $s$ do
        $c \leftarrow$ the smallest solution congruent to $s \pmod{p^h}$ in the range $[-M, M]$
        while $c \leqslant M$ do                                              ▷ Sieving loop
            $A[c] \leftarrow A[c] - \lfloor 1000 \log p \rfloor$          ▷ $\lfloor 1000 \log p \rfloor$ are globally constant values
            $c \leftarrow c + p^h$                                            ▷ The next solution
        end while
    end for
end for
```

we take the pair $(a/q, b/q)$ into account. If the value of b turns out to be negative, we change the signs of both a and b.

3.2 Parallel Index Calculations

The procedure for data-parallel index calculations is shown in Figure 1. An SIMD register is packed with k 32-bit integer or single-precision floating-point values. The SIMD registers are of sizes 128 and 256 bits in SSE2 and AVX, respectively. So we have $k = 4$ or 8. Parallel index calculations proceed in the following way. We pack an SIMD register with the values $s_1, s_2, \ldots, s_k$ and another SIMD register with the values $p_1^{h_1}, p_2^{h_2}, \ldots, p_k^{h_k}$, where $s_1, s_2, \ldots, s_k$ are the solutions of the congruence $T(c) \equiv 0$ modulo $p_1^{h_1}, p_2^{h_2}, \ldots, p_k^{h_k}$, respectively. These initial solutions are taken to be as small as possible in the sieving range. Then, k parallel additions take place with the help of a single SIMD addition instruction. The individual indices are obtained by extracting the components of the SIMD register storing the sum. However, a repacking of these indices is not needed in the next iteration. The output SIMD register of one iteration is directly fed as an input in the next iteration. In all these index calculations, the second SIMD register (holding the p^h values) remains constant. The parallel index-calculation loop terminates when the value of any of the k components goes beyond the sieving range. The smoothness candidates in the NFSM are polynomial expressions in two variables a, b. If we fix b and vary a, sieving becomes identical to the univariate case of $T(c)$. A couple of general points concerning these data-parallel implementations are in order.

In both the SSE2 and AVX implementations, we use 32-bit chunks (integers or floating-point numbers) to populate SIMD registers. This means that a single vector addition in SSE2 computes four sums in parallel. For AVX, eight sums are carried out by each such vector addition. Both SSE2 and AVX support packing of 64-bit units (double-precision integers or floating-point numbers). But this reduces the extent of parallelism. Moreover, so long as array indices are restricted to single-precision values

Fig. 1. SIMD-based index calculations during sieving

$$\begin{array}{|c|}\hline s_1\\\hline s_2\\\hline s_3\\\hline \cdot\\ \cdot\\ \cdot\\\hline s_k\\\hline\end{array} + \begin{array}{|c|}\hline p_1^{h_1}\\\hline p_2^{h_2}\\\hline p_3^{h_3}\\\hline \cdot\\ \cdot\\ \cdot\\\hline p_k^{h_k}\\\hline\end{array} = \begin{array}{|c|}\hline s_1+p_1^{h_1}\\\hline s_2+p_2^{h_2}\\\hline s_3+p_3^{h_3}\\\hline \cdot\\ \cdot\\ \cdot\\\hline s_k+p_k^{h_k}\\\hline\end{array}, \quad \begin{array}{|c|}\hline s_1+p_1^{h_1}\\\hline s_2+p_2^{h_2}\\\hline s_3+p_3^{h_3}\\\hline \cdot\\ \cdot\\ \cdot\\\hline s_k+p_k^{h_k}\\\hline\end{array} + \begin{array}{|c|}\hline p_1^{h_1}\\\hline p_2^{h_2}\\\hline p_3^{h_3}\\\hline \cdot\\ \cdot\\ \cdot\\\hline p_k^{h_k}\\\hline\end{array} = \begin{array}{|c|}\hline s_1+2p_1^{h_1}\\\hline s_2+2p_2^{h_2}\\\hline s_3+2p_3^{h_3}\\\hline \cdot\\ \cdot\\ \cdot\\\hline s_k+2p_k^{h_k}\\\hline\end{array}, \ \cdots.$$

(as is usually the case), use of 64-bit units not only is wasteful of space but also increases the relative overhead associated with packing and unpacking.

AVX provides 256-bit vector operations on floating-point numbers only. Therefore, we transform 32-bit integers to 32-bit single-precision floating-point numbers and pack these floating-point numbers in an SIMD register. After a vector addition, the floating-point units are converted back to integers after unpacking. This, however, comes at a cost. First, this introduces overheads associated with conversion between integer and floating-point formats. Second, this reduces the maximum index value from 2^{32} to 2^{23} (since the length of the mantissa segment of a 32-bit single-precision floating-point number is 23 bits according to the IEEE Floating-Point Standard). Converting integers to double-precision floating-point numbers eliminates this restriction. But as mentioned above, it is preferable to avoid 64-bit units in SIMD registers.

3.3 SSE2 Implementations

SSE2 provides 128-bit SIMD registers, that is, we take $k = 4$ in Fig 1. We take four integer solutions s_i as small as possible in the sieving interval. These four solutions correspond to four $p_i^{h_i}$ values, all of which need not be distinct from one another. For $j \geqslant 1$, four index calculations $s_i + jp_i^{h_i}$ are carried out in parallel by adding $p_i^{h_i}$ to $s_i + (j-1)p_i^{h_i}$. The indices are extracted in each iteration and $\log p_i$ values are subtracted from these array locations. Algorithm 3 summarizes this SSE2-based implementation of the MPQSM sieve in the form of a pseudocode.

The choice of the four $p_i^{h_i}$ values in each parallel sieving loop has a bearing on the running time of the loop. This is mostly relevant to the MPQSM in which the $p_i^{h_i}$ values may differ considerably from one another if the exponents h_i are large. We have implemented only the incomplete sieving for the NFSM and the lattice sieve, so all h_i values are 1, and we can choose the primes p_i close to one another.

For each $p_i^{h_i}$, the number of iterations in the sieving loop is $\lfloor (2M+1)/p_i^{h_i} \rfloor$. We stop data-parallel index calculations as soon as one of the indices $s_i + jp_i^{h_i}$ exceeds the sieving bound. If all the $p_i^{h_i}$ values were the same, then the loop would run for an optimal number of iterations. But, in general, there are only two solutions for each $p_i^{h_i}$ value in the MPQSM, since we are dealing with a quadratic congruence $T(c) \equiv 0 \pmod{p_i^{h_i}}$. Therefore, all the $p_i^{h_i}$ values loaded in the SIMD register cannot be the same. If one of

Algorithm 3 Pseudocode for the implementation of the MPQSM sieve using SSE2

```
Input : Factor base $FB$, sieving bound $M$, sieving array $A$ indexed by $c$
Output: Sieving array $A$ after subtractions of log values

for all $c$ in the range $-M \leqslant c \leqslant M$ do
    $A[c] \leftarrow \lfloor 1000 \log T(c) \rfloor$
end for
for all prime powers $p^h$ and all solutions $s$ of $T(c) \equiv 0 \pmod{p^h}$ do
                                        ▷ Prime $p \in FB$ and the solutions are precomputed
    Divide these solutions in groups of four: Let $s_1, s_2, s_3, s_4$ be such a group
                    ▷ $s_i$ corresponds to the value $p_i^{h_i}$, the four values being not necessarily the same
    for all groups of solution values $s_1, s_2, s_3, s_4$ do
        $c_i \leftarrow$ the smallest solution congruent to $s_i \pmod{p_i^{h_i}}$ in the range $[-M, M]$
        Load four $p_i^{h_i}$ values in SIMD register $R_0$        ▷ Fast 16-byte aligned packing
        Load four $c_i$ values in SIMD register $R_1$        ▷ Fast 16-byte aligned packing
        while all $c_i \leqslant M$ do                                  ▷ Sieving loop
            $A[c_1] \leftarrow A[c_1] - \lfloor 1000 \log p_1 \rfloor$
            $A[c_2] \leftarrow A[c_2] - \lfloor 1000 \log p_2 \rfloor$              ▷ The log values are
            $A[c_3] \leftarrow A[c_3] - \lfloor 1000 \log p_3 \rfloor$              ▷ globally constant
            $A[c_4] \leftarrow A[c_4] - \lfloor 1000 \log p_4 \rfloor$
            $R_1 \leftarrow R_0 + R_1$                ▷ Four additions replaced by one vector operation
            Unpack $c_1, c_2, c_3, c_4$ from $R_1$              ▷ Fast 16-byte aligned unpacking
        end while
        for $i = 1$ to 4 do                      ▷ Handle the leftover values sequentially
            while $c_i \leqslant M$ do
                $A[c_i] \leftarrow A[c_i] - \lfloor 1000 \log p_i \rfloor$
                $c_i \leftarrow c_i + p_i^{h_i}$
            end while
        end for
    end for
end for
```

the $p_i^{h_i}$ values is significantly larger than the other components in the SIMD register, the index calculations for the smaller $p_i^{h_i}$ values suffer from a premature termination in the parallel SIMD loop. Moreover, widely different $p_i^{h_i}$ values endanger spatial proximity during log-value subtractions, leading to an increased number of cache misses.

If we take the same p and use different h_i values in the components of the SIMD register, the problem just mentioned becomes quite acute. It is instead preferable to use the same or close p_i values and take the same h value in all the components of the SIMD register. This means that we first sieve for all the solutions of $T(c) \equiv 0 \pmod{p_i}$ for $i = 1, 2, 3, \ldots, t$ in groups of four. Next, we take $h = 2$ and do sieving for the solutions of $T(c) \equiv 0 \pmod{p_i^2}$, again in groups of four, and so on. For this choice, the iteration counts $\lfloor (2M+1)/p_i^{h_i} \rfloor$ are nearly the same for all the four components, and the extent of parallelism increases. Moreover, the spatial proximity during the access of A improves for small values of p and h. Notice that our NFSM and lattice sieve implementations use $h = 1$ only, and therefore we only need to take the p_i values in the four components as close to one another as possible (like consecutive primes).

We have used some other optimization tricks. If $T(c) \equiv 0 \pmod{p^h}$ has only a few solutions in the sieving interval, then it is preferable to sieve for these values sequentially, since in this case the benefits of parallelization is shadowed by packing and unpacking overheads. The threshold, up to which parallelizing solutions modulo p^h remains beneficial, is determined experimentally.

The MPQSM deals with a quadratic congruence $T(c) \equiv 0 \pmod{p^h}$ which usually has two solutions (if p is odd). If there is a unique solution of this congruence (this is rather infrequent), we handle the sieving for this solution sequentially in order not to lose the alignment of using a pair of two distinct prime powers in a round of data-parallel sieving. For the NFSM and the lattice sieve, we solve a linear congruence to generate the initial solutions. In this case, there is always a unique initial solution s_i for each p_i (or for each root of $f(x)$ modulo p_i in the algebraic sieve of the NFSM). Here, we carry out sieving for four solutions in parallel.

Intrinsics Used for SSE2 The SSE2 intrinsics used in our implementation are shown in Figure 2.[2] The header file `emmintrin.h` defines the 128-bit data type `__m128i` and the prototypes for intrinsics `_mm_load_si128`, `_mm_add_epi32` and `_mm_store_si128`. The registers `xmm_p` (for p^h or p values) and `xmm_l` (for s values) are packed each with four contiguous 32-bit integers starting from the locations `P` and `L`, respectively, using `_mm_load_si128`. Then, they are added with a single vector instruction defined by `_mm_add_epi32`. Finally, the output SIMD register `xmm_l` is unpacked and its content is stored in the location `L`. However, unpacking is not destructive, that is, we can reuse this output register later, if required. Now, we subtract the log values from the array locations stored in `L[0]`, `L[1]`, `L[2]` and `L[3]`. To use the intrinsics `_mm_load_si128` and `_mm_store_si128`, it is necessary that the addresses `P` and `L` are 16-byte aligned. If they are not, we have to use the intrinsics `_mm_loadu_si128` and `_mm_storeu_si128`. However, these intrinsics are more time-consuming compared to the ones for 16-byte

[2] Only the SIMD intrinsics are shown in the figure. The first two intrinsics are used before the sieving loop, whereas the last two intrinsics are used in each iteration of the sieving loop. The same comments apply to Figure 3 as well.

Fig. 2. SSE2 intrinsics used for index calculations

```
__m128i xmm_p = _mm_load_si128 (__m128i *P);
__m128i xmm_l = _mm_load_si128 (__m128i *L);
__m128i xmm_l = _mm_add_epi32 (__m128i xmm_l, __m128i
xmm_p);
_mm_store_si128 (__m128i *L, __m128i xmm_l);
```

Fig. 3. AVX intrinsics used for index calculations

```
__m256 ymm_p = _mm256_load_ps (float *P);
__m256 ymm_l = _mm256_load_ps (float *L);
__m256 ymm_l = _mm256_add_ps (__m256 ymm_l, __m256
ymm_p);
_mm256_store_ps (float *L, __m256 ymm_l);
```

aligned memory. Another important point is that the packing overhead is much larger in case we attempt to pack from four non-contiguous locations using `_mm_set_epi32` or similar intrinsics. So, we avoid them in our implementations.

3.4 AVX Implementations

Following the same basic idea discussed above for SSE2, we implement data-parallel index calculations using AVX intrinsics. The AVX instruction set of Sandy Bridge does not support 256-bit vector integer operations. In order to exploit the power of 256-bit registers, we carry out floating-point index calculations. But then, we also need conversions between floating-point numbers and integers, since array indices must be integers.

Intrinsics Used for AVX The intrinsics we employ in our AVX implementation are shown in Figure 3. The 256-bit data type `__m256` and the prototypes for the intrinsics `_mm256_load_ps`, `_mm256_add_ps` and `_mm256_store_ps` are defined in the header file `immintrin.h`. Two 256-bit SIMD registers (`ymm_p` and `ymm_l`) are packed each with eight contiguous 32-bit floating-point numbers starting from the locations `P` (for p^h or p values) and `L` (for s values), respectively, using `_mm256_load_ps`. A single vector instruction defined by `_mm256_add_ps` is used to add them. The individual results in the output SIMD register `ymm_l` are then extracted in the location starting from the address `L`. Now, we need to convert the floating-point values `L[0]`, `L[1]`, `L[2]`, `L[3]`, `L[4]`, `L[5]`, `L[6]` and `L[7]` to integers to obtain the array locations for sieving. The addresses `P` and `L` should be 32-byte aligned, for otherwise we need to use the inefficient intrinsics `_mm256_loadu_ps` and `_mm256_storeu_ps`. Moreover, we avoid using `_mm256_set_ps` or similar inefficient intrinsics which are used to pack eight floating-point numbers from arbitrary non-contiguous locations.

4 Experimental Results and Analysis

4.1 Experimental Setup

Our focus is on implementing the sieving part efficiently using SIMD parallelization. Version 4.6.3 of GCC supports SSE2 and AVX intrinsics. Our implementation platform is a 2.40GHz Intel Xeon machine (Sandy Bridge microarchitecture with CPU Number E5-2609). Version 2.5.0 of the GP/PARI calculator (see [25]) is used to calculate the log values of large integers and to find the zeros of f modulo p (for only the algebraic sieve in the NFSM). We use the optimization flag `-O3` with GCC for all sequential and parallel implementations. To avoid the AVX-SSE and SSE-AVX conversions, we use the flag `-mavx` in the AVX implementation. To handle large integers and operations on them, we use version 5.0.5 of the library [26].

4.2 Speeding up Implementations of the Sieving Using SSE2 and AVX

MPQSM Table 1 summarizes the timing (in milliseconds) and speedup figures for the implementations of the MPQSM sieve. For each n, M, B values used in our experiments, we take the average of the times taken by fifty executions. In all the tables, the speedup figures are calculated with respect to the sequential implementations. We have incorporated all the improvement possibilities discussed in Section 3.3. The rows in the same cluster have the same values for M and B, but differ in the count of digits in the integer being factored.

NFSM Timings for our implementations are reported in milliseconds. For each data set, we record the average of the times taken over fifty executions. We take $B_{rat} = B_{alg} = B'$ as the bounds on the small primes in the two sieves. We document the results for $1 \leqslant b \leqslant 10$. Timing and speedup figures for the implementations (sequential and SIMD-based) are summarized in Table 2 and Table 3 for the 60- and 120-digit integers n_1 and n_2, respectively (see Section 3). The experimental results for the rational sieve and the algebraic sieve are shown separately.

Lattice Sieve Method We have implemented the rational sieve of the NFSM using the lattice sieve method. For each data set, we record the average of the times taken by fifty executions. Table 4 and Table 5 show the experimental results for the 60- and 120-digit integers n_1 and n_2, respectively (see Section 3), with $B_{rat} = 10000$ and $L_2 = 10$. Different values of B'_{rat} are considered. Times are measured in seconds.

Observations Tables 1–5 suggest the following points.

– For the MPQSM sieve, the speedup varies in the range 20–35% on an average, except in the last two clusters where both M and B values are large.
– For the NFSM sieve, we get a speedup in the range 15–40%.

Table 1. Timing and speedup figures of our implementations of the MPQSM sieve

Number of digits in n	Sieving limit M	Bound on small primes B	Sequential Time (in ms)	SSE2 Parallelization Time (in ms)	SSE2 Parallelization Speedup (in %)	AVX Parallelization Time (in ms)	AVX Parallelization Speedup (in %)
39	500000	46340	9.14	7.00	23.38	7.02	23.17
100	500000	46340	10.66	8.40	21.22	8.45	20.74
152	500000	46340	15.21	10.65	29.99	11.46	24.68
247	500000	46340	10.34	7.84	24.24	7.97	22.97
89	2000000	46340	69.37	49.54	28.59	49.86	28.12
187	2000000	46340	76.51	49.67	35.08	50.08	34.55
247	2000000	46340	85.59	56.19	34.35	58.31	31.88
93	5000000	46340	319.63	216.38	32.30	228.93	28.38
152	5000000	46340	398.74	260.60	34.64	262.27	34.22
241	5000000	46340	196.84	156.62	20.44	160.11	18.66
65	3000000	300000	120.36	92.22	23.38	94.57	21.43
158	3000000	300000	206.41	124.75	39.56	124.88	39.50
241	3000000	300000	115.81	91.59	20.91	92.35	20.26
100	5000000	463400	333.82	265.41	20.49	259.05	22.40
187	5000000	463400	258.99	193.09	25.45	194.93	24.74
241	5000000	463400	217.96	177.25	18.67	179.33	17.72
65	5000000	803400	231.10	187.93	18.68	189.14	18.16
158	5000000	803400	370.68	264.92	28.53	256.66	30.76
247	5000000	803400	295.03	253.97	13.92	256.43	13.08
65	4000000	4000000	211.23	183.36	13.19	192.86	8.70
187	4000000	4000000	248.04	207.85	16.21	208.36	16.00
251	4000000	4000000	260.76	211.87	18.75	212.01	18.70
65	5000000	5000000	274.38	242.38	11.66	244.66	10.83
158	5000000	5000000	370.98	312.51	15.76	312.53	15.76
241	5000000	5000000	256.53	238.41	7.06	240.24	6.35

Table 2. Timing and speedup figures of our implementations of the NFSM sieve for n_1

	Sieving limit u	Bound on small primes B'	Sequential Time (in ms)	SSE2 Parallelization Time (in ms)	SSE2 Parallelization Speedup (in %)	AVX Parallelization Time (in ms)	AVX Parallelization Speedup (in %)
Rational Sieve	500000	50000	95.59	79.40	16.93	79.11	17.24
	3000000	50000	1677.30	1208.97	27.92	1206.97	28.04
	3000000	300000	1816.98	1358.18	25.25	1354.13	25.47
Algebraic Sieve	500000	50000	90.49	71.69	20.78	71.66	20.81
	3000000	50000	1564.31	1116.66	28.62	1118.55	28.50
	3000000	300000	1700.84	1266.34	25.55	1260.98	25.86

Table 3. Timing and speedup figures of our implementations of the NFSM sieve for n_2

	Sieving limit u	Bound on small primes B'	Sequential Time (in ms)	SSE2 Parallelization Time (in ms)	SSE2 Parallelization Speedup (in %)	AVX Parallelization Time (in ms)	AVX Parallelization Speedup (in %)
Rational Sieve	600000	60000	126.32	100.02	20.82	101.59	19.58
	2000000	60000	970.24	607.13	37.42	605.07	37.64
	2000000	200000	993.67	666.97	32.88	663.11	33.27
Algebraic Sieve	600000	60000	111.50	83.10	25.47	83.52	25.10
	2000000	60000	866.23	523.22	39.60	527.01	39.16
	2000000	200000	912.86	581.89	36.26	579.06	36.57

Table 4. Timing and speedup figures of our implementations of the rational sieve of the NFSM using the lattice sieve method for n_1

Ratio B'_{rat}/B_{rat}	Sieving limit L_1	Sequential Time (in seconds)	SSE2 Parallelization Time (in seconds)	SSE2 Parallelization Speedup (in %)	AVX Parallelization Time (in seconds)	AVX Parallelization Speedup (in %)
0.1	100000	11.72	10.98	6.31	10.98	6.32
	500000	96.81	77.66	19.78	77.53	19.92
	800000	164.42	126.59	23.00	126.98	22.77
0.2	100000	10.51	9.73	7.40	9.78	6.88
	500000	86.64	68.86	20.52	69.87	19.35
	800000	148.50	112.63	24.15	112.71	24.10
0.3	100000	9.14	8.50	6.97	8.53	6.64
	500000	75.16	60.10	20.04	60.06	20.09
	800000	129.79	98.70	23.95	98.67	23.98
0.4	100000	7.85	7.30	6.95	7.30	6.93
	500000	64.54	51.55	20.13	51.53	20.16
	800000	111.79	84.47	24.43	84.71	24.23
0.5	100000	6.54	6.14	6.11	6.14	6.07
	500000	53.55	42.84	20.01	42.83	20.02
	800000	94.49	70.40	25.50	70.30	25.60

Table 5. Timing and speedup figures of our implementations of the rational sieve of the NFSM using the lattice sieve method for n_2

Ratio B'_{rat}/B_{rat}	Sieving limit L_1	Sequential Time (in seconds)	SSE2 Parallelization		AVX Parallelization	
			Time (in seconds)	Speedup (in %)	Time (in seconds)	Speedup (in %)
0.1	100000	11.91	11.04	7.32	11.04	7.32
	500000	95.95	76.46	20.32	76.42	20.35
	800000	167.25	125.34	25.06	127.78	23.60
0.2	100000	10.57	9.76	7.67	9.77	7.57
	500000	85.20	67.66	20.59	67.62	20.63
	800000	148.42	110.92	25.26	110.75	25.38
0.3	100000	9.18	8.49	7.50	8.52	7.23
	500000	75.33	59.99	20.37	60.00	20.35
	800000	129.66	99.08	23.59	98.87	23.75
0.4	100000	7.85	7.28	7.26	7.29	7.06
	500000	63.71	50.55	20.66	50.55	20.66
	800000	110.06	82.92	24.65	82.86	24.71
0.5	100000	6.55	6.05	7.64	6.05	7.55
	500000	52.98	42.05	20.63	42.07	20.58
	800000	92.53	69.03	25.40	71.22	23.03

– The speedup obtained in the lattice sieve (in the range 5–25%) is less compared to what is achieved by the line sieve. This is attributed to the fact that the lattice sieve involves some extra work like resetting the sieving array and calculating a reduced basis for each medium prime $q \in M$.

– The speedup increases when the sieving limit increases. This is due to fact that larger sieving bounds allow parallel index calculations to proceed for a larger number of iterations.

– We get higher speedup for smaller bounds on the primes in the factor base. In this case too, the number of iterations in the sieving loop increases.

– Because of frequent conversions between integer and floating-point formats in each iteration of the sieving loop, the speedup with AVX, despite the use of 256-bit registers, is below our expectation, and turns out to be almost the same as that with SSE2.

– For the NFSM, the speedup is found to be somewhat higher for the algebraic sieve compared to the rational sieve. The rational sieve packs four (or eight) different primes in a SIMD register. The largest of these determines how many times the loop iterates. In the algebraic sieve, the primes packed in the SIMD register may be repeated, since the polynomial f may have multiple roots modulo a small prime. Repeated primes indicate the possibility of an increased number of iterations in the sieving loop. This is the most likely reason why we obtain better speedup in the algebraic sieve.

Table 6. Cache-profiling figures for the MPQSM sieve using SSE2

Sieving limit M	Bound on small primes B	Sum of L1 cache misses		Sum of L3 cache misses	
		Basic Version	Improved Version	Basic Version	Improved Version
3000000	300000	8777554	8044299	5631142	4997079
5000000	46340	12968558	11980434	10656421	9135800
5000000	463400	14957182	13995977	12145165	10623400

4.3 Cache-Profile Analysis of Our Implementations of the MPQSM Sieve Using SSE2

The idea of packing close $p_i^{h_i}$ values in SIMD registers, as discussed in Section 3.3, was motivated by the heuristic assumption that the improvement makes our implementations more cache-friendly than the basic one (which corresponds to the same p_i value and different h_i values). In order to verify the effectiveness of our cache-friendly implementation, we have done cache-profile analysis our implementations (both the basic and the improved versions) using SSE2.

We have used the Cachegrind tool along with Valgrind (Version 3.7.0) to profile our basic and improved SIMD implementations using SSE2. From those profile information, we get an idea about the L1 and LL (last level, L3 in this case) cache misses. The sum of L1 misses includes L1 instruction-fetch misses, L1 data-read misses, and L1 data-write misses. Similarly, the sum of L3 misses includes L3 instruction-fetch misses, L3 data-read misses, and L3 data-write misses. The sum of L1 cache misses and the sum of L3 cache misses for an integer with 241 digits for different M and B values are reported in Table 6. The table clearly indicates that our cache-friendly implementation encounters about 10% less cache misses compared to the unoptimized version.

5 Conclusion

Sieving is used to make the relation-collection phase in modern integer-factoring algorithms faster than trial divisions. In this work, we have exploited SIMD features, commonly available in modern microprocessors, in our implementations of the sieving procedure. We have chosen the two most efficient factoring algorithms (the MPQSM and the NFSM) in our experiments. We have implemented both the line sieve and the lattice sieve. We have used SSE2 and AVX features provided by Intel's Sandy Bridge architecture. Our data-parallel implementations have achieved noticeable speedup over sequential implementations. Line sieving enjoys slightly higher speedup than lattice sieving. We have also employed some optimization techniques in our SIMD-based implementations, and these have been experimentally verified to increase the cache-friendliness of our implementations. We conclude this paper after mentioning some directions in which this research can be extended.

– Although we have been able to speed up index calculations using SIMD-based parallelization, a similar approach was not effective during data-parallel subtractions

Fig. 4. SIMD-based subtraction of log values during sieving

$$\begin{array}{|c|}\hline A[s_1]\\\hline A[s_2]\\\hline A[s_3]\\\hline A[s_4]\\\hline\end{array} - \begin{array}{|c|}\hline \log p_1\\\hline \log p_2\\\hline \log p_3\\\hline \log p_4\\\hline\end{array} = \begin{array}{|c|}\hline A[s_1]-\log p_1\\\hline A[s_2]-\log p_2\\\hline A[s_3]-\log p_3\\\hline A[s_4]-\log p_4\\\hline\end{array},$$

$$\begin{array}{|c|}\hline A[s_1+p_1^{h_1}]\\\hline A[s_2+p_2^{h_2}]\\\hline A[s_3+p_3^{h_3}]\\\hline A[s_4+p_4^{h_4}]\\\hline\end{array} - \begin{array}{|c|}\hline \log p_1\\\hline \log p_2\\\hline \log p_3\\\hline \log p_4\\\hline\end{array} = \begin{array}{|c|}\hline A[s_1+p_1^{h_1}]-\log p_1\\\hline A[s_2+p_2^{h_2}]-\log p_2\\\hline A[s_3+p_3^{h_3}]-\log p_3\\\hline A[s_4+p_4^{h_4}]-\log p_4\\\hline\end{array}, \cdots.$$

of log values. Our efforts on parallelizing the subtraction operations, as described in Figure 4, have not produced any benefit. Here, the bottleneck is that we cannot reuse the content of the output SIMD register (used for storing array elements) in the next iteration. Consequently, packing is required in every iteration of the sieving loop. Moreover, both packing and unpacking access non-contiguous memory locations, leading to additional slowdown in the implementations.

Efficient data parallelization of the subtraction operations seems to be quite challenging, since a straightforward use of SIMD intrinsics is not beneficial, as mentioned above. The main bulk in the sieving computations includes index calculations and subtractions of log values. It requires non-trivial experimental investigations to settle whether subtractions can at all be effectively handled by SIMD intrinsics.

- Our implementations using 256-bit SIMD registers can be easily ported to Intel's recently released Haswell architecture, where 256-bit vector integer instructions are available. Using this AVX2 instruction set helps us to avoid frequent conversions between floating-point numbers and integers. This has the potential to increase the performance gains significantly.
- Table 1 for sieving in the MPQSM shows that the speedup is somewhat small when both M and B are large. Improving the performance of our SIMD-based implementations for large values of M and B deserves further attention.
- For the lattice sieve method, Pollard [10] proposes two different variants: sieving by rows and sieving by vectors. In our work, we have dealt with the first variant only. It is an interesting experimental investigation to determine how SIMD features can benefit sieving by vectors.
- Our implementations of the MPQSM and NFSM sieves are not readily portable to polynomial sieves used in the computation of discrete logarithms over finite fields of small characteristics (for example, see [27, 28]). Fresh experimentation is needed to investigate the effects of SIMD parallelization on polynomial sieves. Data-parallel implementations of the pinpointing algorithm of Joux [29] also merits attention.

References

1. Dixon, B., Lenstra, A.K.: Factoring integers using SIMD sieves. In: EUROCRYPT. (1993) 28–39
2. Bernstein, D.J., Lenstra, A.K.: A general number field sieve implementation. In: The Development of the Number Field Sieve, Lecture Notes in Mathematics, vol. 1554. (1993) 103–126
3. Dixon, J.D.: Asymptotically fast factorization of integers. Mathematics of Computation **36** (1981) 255–260
4. Lehmer, D.H., Powers, R.E.: On factoring large numbers. Bulletin of the American Mathematical Society **37** (1931) 770–776
5. Morrison, M.A., Brillhart, J.: A method of factoring and the factorization of F_7. Mathematics of Computation **29** (1975) 183–205
6. Pomerance, C.: The quadratic sieve factoring algorithm. In: EUROCRYPT. (1984) 169–182
7. Silverman, R.D.: The multiple polynomial quadratic sieve. Mathematics of Computation **48** (1987) 329–339
8. Lenstra, A.K., Lenstra, H.W., Manasse, M.S., Pollard, J.M.: The number field sieve. In: STOC. (1990) 564–572
9. Buhler, J.P., Lenstra, H.W., Pomerance, C.: Factoring integers with the number field sieve. In: The Development of the Number Field Sieve, Lecture Notes in Mathematics, vol. 1554. (1993) 50–94
10. Pollard, J.M.: The lattice sieve. In: The Development of the Number Field Sieve, Lecture Notes in Mathematics, vol. 1554. (1993) 43–49
11. Coppersmith, D., Odlyzko, A.M., Schroeppel, R.: Discrete logarithms in GF(p). Algorithmica **1**(1) (1986) 1–15
12. Lenstra, H.W.: Factoring integers with elliptic curves. Annals of Mathematics **126** (1987) 649–673
13. Montgomery, P.L.: A block Lanczos algorithm for finding dependencies over GF(2). In: EUROCRYPT. (1995) 106–120
14. Intel Corporation: Intrinsics for Intel(R) Streaming SIMD Extensions (2011) `http://scc.qibebt.cas.cn/docs/compiler/intel/2011/compiler_c/main_cls/intref_cls/common/intref_bk_sse.htm`.
15. Lomont, C.: Introduction to Intel® Advanced Vector Extensions. `http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions/` (2011)
16. Konsor, P.: Avoiding AVX-SSE transition penalties (2012) `http://software.intel.com/en-us/articles/avoiding-avx-sse-transition-penalties/`.
17. Shanks, D.: Five number-theoretic algorithms. In: Proceedings of the Second Manitoba Conference on Numerical Mathematics. (1973) 51–70
18. Tonelli, A.: Bemerkung über die auflösung quadratischer congruenzen. Göttinger Nachrichten (1891) 344–346
19. Das, A.: Computational Number Theory (Discrete Mathematics and Its Applications). Chapman and Hall/CRC (2013)
20. Davis, J.A., Holdridge, D.B.: Factorization using the quadratic sieve algorithm. In: CRYPTO. (1983) 103–113
21. Etienne, H.: LLL lattice basis reduction algorithm (March 2010) `http://algo.epfl.ch/_media/en/projects/bachelor_semester/rapportetiennehelfer.pdf`.
22. Nguyen, P.Q., Stehlé, D.: Low-dimensional lattice basis reduction revisited. In: ANTS. (2004) 338–357

23. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H.J.J., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In: CRYPTO. (2010) 333–350
24. Briggs, M.E.: An introduction to the general number field sieve. Master's thesis, Virginia Polytechnic Institute and State University (1998) `https://www.math.vt.edu/people/brown/doc/briggs_gnfs_thesis.pdf`.
25. Cohen, H., Belabas, K.: PARI/GP home (2003–2016) `http://pari.math.u-bordeaux.fr/`.
26. GMP: The GNU Multiple Precision Arithmetic Library (2000–2016) `http://gmplib.org/`.
27. Adleman, L.M., Huang, M.D.A.: Function field sieve method for discrete logarithms over finite fields. Information and Computation **151**(1-2) (1999) 5–16
28. Gordon, D.M., McCurley, K.S.: Massively parallel computation of discrete logarithms. In: CRYPTO. (1992) 312–323
29. Joux, A.: Faster index calculus for the medium prime case application to 1175-bit and 1425-bit finite fields. In: EUROCRYPT. (2013) 177–193