SIMD-Based Implementations of 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. The best known integer-factoring algorithms consist of two stages: the sieving stage and the linear-algebra stage. Efficient parallel implementations of both these stages have been reported in the literature. All these implementations are based on multi-core or distributed parallelization. In this paper, we experimentally demonstrate that SIMD instructions available in many modern processors can lead to additional speedup in the computation of each core. We handle the sieving stage of the two fastest known factoring algorithms (NFSM and MPQSM), and are able to achieve 15–40% speedup over non-SIMD implementations. Although the sieving stage offers many tantalizing possibilities of data parallelism, exploiting these possibilities to get practical advantages is a challenging task. Indeed, to the best of our knowledge, no similar SIMD-based implementation of sieving seems to have been reported in the literature.

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

1 Introduction

Factoring large integers has been of much importance in cryptography and computational number theory. Many cryptosystems like RSA derive their security from the apparent intractability of factoring large integers. Indeed, the integer factorization problem can be dubbed as the fundamental computational problem in number theory. Despite many attempts to solve this problem efficiently, researchers could not come up with any polynomial-time algorithm so far. Further studies of factoring and efficient implementation issues continue to remain an important area of research of both practical and theoretical significance.

Given a composite integer n, the integer factorization problem can be formally framed as to find out all the prime divisors $p_1, p_2, \ldots, p_l$ of n and their corresponding multiplicities $v_{p_1}, v_{p_2}, \ldots, v_{p_l}$, where

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

B. Gierlichs, S. Guilley, and D. Mukhopadhyay (Eds.): SPACE 2013, LNCS 8204, pp. 40–55, 2013.

Many algorithms are proposed to solve the integer factorization problem. The older algorithms have running times exponential in the input size (the number of bits in n, that is, $\log_2 n$ or $\lg n$). The time and space complexities of the modern integer-factoring algorithms are subexponential in $\lg n$. These algorithms typically consist of two stages. In the first stage, a large number of candidates are generated. Each such candidate that factors completely over a factor base, a set of small primes, yields a relation. In the second stage, the relations obtained in the first stage are combined, by solving a set of linear equations modulo 2, to get a congruence 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.

The sieving stage is introduced in the quadratic sieve method [1] in an attempt to make the relation-collection stage more efficient compared to trial division. Let p be small prime, and g an integer-valued polynomial function of a variable c such that $g(c) \equiv x \pmod{p}$ for some $x \in \mathbb{Z}$. If γ is a solution of the congruence, then all of its solutions are of the form $c = \gamma + kp$, where $k \in \mathbb{Z}$. This is how trial divisions are avoided in sieving.

SIMD (Single Instruction Multiple Data)-based architecture is a recent technology that comes with *vector* instructions and register sets. The size of these special SIMD registers is larger than (usually a multiple of) that of the general-purpose registers. Multiple data of the same type can be accommodated in an SIMD register. This is frequently called *packing*. A binary operation on two packed registers can be performed using a single vector instruction. The individual results are extracted from the output SIMD register. This extraction process is known as *unpacking*. The less the packing and unpacking overheads are, the more are the advantages that can be derived from SIMD-based parallelization.

The sieving part turns out to be the most time-consuming stage in factoring algorithms. As a result, efficient implementations of the sieving stage is of the utmost importance in the context of factoring algorithms. Sieving is, however, massively parallelizable on multi-core and even on distributed platforms. Our work is not an attempt to exploit parallelism in the multi-core or distributed level. On the contrary, we attempt to investigate how SIMD-based parallelism can provide additional speedup within each single core. In sieving, both index calculations and subtractions of log values involve data-parallel operations. SIMD intrinsics have the potential of increasing the efficiency of both these steps. Unfortunately, frequent packing and unpacking of data between regular registers (or memory locations) and SIMD registers stand in the way of this potential benefit of data parallelism. Our major challenge in this work is to reduce the packing and unpacking overheads. So far, we have been able to achieve some speedup in the index-calculation process. Achieving similar speedup in the subtraction phase still eludes us. To the best of our knowledge, no SIMD-based parallelization attempts on sieving algorithms for integer factorization are reported in the literature. The early implementations described in [2,3] use the term SIMD but are akin to multi-core parallelization in a 16K MasPar SIMD machine with 128×128 array of processing elements.

Our Contribution: In this paper, we concentrate on efficient implementations of sieving using Intel's SSE2 (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) features. We handle the sieving stages of two factoring algorithms stated below. Currently, these are the most practical factoring algorithms.

- The multiple polynomial quadratic sieve method (MPQSM) uses the same concept as the quadratic sieve method (QSM), except that the MPQSM works with a general polynomial instead of a fixed one. By varying the coefficients of this general polynomial, we generate different instances of sieving, which can run in parallel, independent of one another. We assume that on a core the polynomial remains fixed, and a sieving interval is provided to us. We use SIMD operations to sieve this interval for the given polynomial. The MPQSM is widely accepted as the second fastest factoring algorithm.
- The general number field sieve method (NFSM) is the fastest known algorithm for factoring integers, and is based upon the theory of algebraic number fields. In particular, a number ring $\mathfrak{O}$ and a homomorphism $\mathfrak{O} \to \mathbb{Z}_n$ are used. The sieving procedure is carried out in both the rings (algebraic and rational sieving). We apply SIMD parallelization to both these sieves.

The rest of the paper is organized as follows. In Section 2, we briefly discuss the background needed for the following sections. This includes a description of the sieving procedures in the MPQSM and in the NFSM, and also of Intel's SSE2 and AVX components. Section 3 illustrates our implementation details for the sieving in the MPQSM and the NFSM. In Section 4, we present our experimental results, and analyze the speedup obtained in our experiments. In the concluding Section 5, we provide some ways in which this work can be extended.

2 Background

2.1 A Summary of Known Integer-Factoring Algorithms

Modern integer-factoring algorithms typically aim to find a congruence 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.

J. D. Dixon [4] proposes the simplest variant of such a factoring method. Based on the work of Lehmer and Powers [5], Morrison and Brillhart introduce another variant known as the CFRAC method [6], where relations are obtained from the continued fraction expansion of $\sqrt{n}$.

In Pomerance's quadratic sieve method (QSM) [1], 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 $T(c)$ 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 QSM, 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 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.

2.2 SSE2 and AVX

SSE (Streaming SIMD Extensions) is an extension of the previous x86 instruction set, and SSE2, introduced in Pentium 4, enhances the SSE instruction set further. This architecture comes with some 128-bit SIMD registers (XMM). In these registers, 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). 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), introduced in Intel's Sandy Bridge processor, is a recent extension to the general x86 instruction set. 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. The AVX instruction set is currently applicable for only floating-point operations.

Programming languages come with intrinsics for high-level access to SIMD instructions both for SSE2 [14] and AVX [15]. We can use these intrinsics directly in our implementations to exploit data parallelism.

2.3 MPQSM

The multiple polynomial quadratic sieve method (MPQSM) [7] is a variant of the quadratic sieve method (QSM) [1]. Instead of using a single polynomial (with fixed coefficients), the MPQSM deals with a general polynomial, and tune its coefficients to generate small smoothness candidates. This variant is parallelizable in the sense that different polynomials can be assigned to different cores. Our aim is to speed up each multi-core implementation, so we work with one of these polynomials. We have implemented the sieving part in the MPQSM using SIMD intrinsics mentioned above. Relations are collected in the MPQSM as follows.

Let us consider a polynomial

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

with $V^2 - UW = n$. We search for the smooth values of $T(c)$, where c can take integer values in the interval $[-M, M]$. W is selected as a prime close to $\frac{\sqrt{2n}}{M}$, such that n is a quadratic residue modulo W. V is the smaller square root of n modulo W, and we take $U = \frac{V^2-n}{W}$ (which is also an integer). Multiplying Eqn (1) by W, we get $WT(c) = (Wc + V)^2 + (UW - V^2) = (Wc + V)^2 - n$, which in turn gives

$$(Wc + V)^2 \equiv WT(c) \pmod{n}. \tag{2}$$

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 needed, 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, we can write Eqn (2) as

$$(Wc + V)^2 \equiv W p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t} \pmod{n}. \tag{3}$$

We include W itself in the factor base. After many such relations like Eqn (3) are collected, we combine those relations to obtain a congruence of the form $x^2 \equiv y^2 \pmod{n}$.

It is required that the number of relations obtained be larger than the number of primes present in the factor base. The advantage of the MPQSM over the QSM is that the coefficients U, V, W of the polynomial are not fixed. We can choose a different prime close to $\frac{\sqrt{2n}}{M}$ as W, modulo which n is a quadratic residue. For a given W, we get fixed values of V and U. Thus, by varying the coefficients, we can generate more relations, keeping M and B fixed.

The methods proposed earlier than QSM use trial divisions to find the B-smooth values of smoothness candidates. QSM introduces a technique called sieving to locate the smooth values using additions/subtractions instead of divisions (along with some preprocessing).

Now, we discuss sieving in the MPQSM briefly. In the MPQSM, we have different values of $T(c)$ for different c in $[-M, M]$, as shown in Eqn (1). We have to locate those 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 $\lfloor 1000 \log |T(c)| \rfloor$ in $A[c]$.

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. Thus, we have to solve the congruence

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

which implies

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

For $h = 1$, we use a root-finding algorithm to compute the square roots of n modulo p. For $h > 1$, we obtain the solutions modulo p^h by lifting the solutions modulo p^{h-1}.

Let s be a solution of $T(c) \equiv 0 \pmod{p^h}$. Then, all the solutions of $T(c) \equiv 0 \pmod{p^h}$ are $s \pm kp^h$, $k \in \mathbb{N}$. Therefore, we subtract $\lfloor 1000 \log p \rfloor$ from all the array locations $A[c]$ such that $c = s \pm kp^h$.

When all such powers of small primes in the factor base are tried out, 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 $\geqslant \lfloor 1000 \log p_{t+1} \rfloor$.

2.4 NFSM

The relation-collection phase of the number field sieve method (NFSM) [9] and some mathematical background are described below.

NFSM 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.

Theorem 2.4.1: If $\theta \in \mathbb{C}$ is a root of a monic irreducible polynomial $f(x)$ of degree d with rational coefficients, then the set of all algebraic integers in $\mathbb{Q}(\theta)$, denoted by $\mathfrak{O}$, forms a subring of the field $\mathbb{Q}(\theta)$.

Theorem 2.4.2: If $\theta \in \mathbb{C}$ is a root of a monic irreducible polynomial $f(x)$ of degree d with integral coefficients, then the set of all $\mathbb{Z}$-linear combinations of the elements $1, \theta, \theta^2, \ldots, \theta^{d-1}$, denoted by $\mathbb{Z}[\theta]$, is a subring of $\mathfrak{O}$.

Theorem 2.4.3: 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 $\phi(1) = 1 \pmod{n}$).

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

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

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

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

$$x^2 \equiv \phi(\alpha)^2 \equiv \phi(\alpha^2) \equiv \phi(\prod_{(a,b)\in S}(a+b\theta))$$
$$\equiv \prod_{(a,b)\in S}\phi(a+b\theta) \equiv \prod_{(a,b)\in S}(a+bm) \equiv y^2 \pmod{n}.$$

If $x \not\equiv \pm y \pmod{n}$, then $\gcd(x-y, n)$ is a non-trivial factor of n.

The NFSM involves a rational factor base and an algebraic factor base. The rational factor base (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 algebraic factor base (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, that is, higher powers of factor-base primes are not considered in sieving.

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 linear 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. We avoid floating-point operations by storing $\lfloor 1000 \log|T(a, b)| \rfloor$ in $A[a, b]$.

After this initialization, we try to find solutions of the congruence $T(a, b) \equiv 0 \pmod{p}$, where p is a small prime in RFB. For a fixed b, the solutions are $a \equiv -bm \pmod{p}$. Let γ be a solution of $T(a, b) \equiv 0 \pmod{p}$ for a particular b. Then, all the solutions of $T(a, b) \equiv 0 \pmod{p}$ for that b are $\gamma \pm kp$, $k \in \mathbb{N}$. Therefore, we subtract $\lfloor 1000 \log p \rfloor$ from all the array locations $A[a, b]$ such that $a = \gamma \pm kp$. We repeat this procedure for all small primes p in the RFB and for all allowed values of b. After this, the array locations storing $A[a, b] \approx 0$ correspond to the smooth values of $T(a, b)$. We apply trial division on these smooth values.

The algebraic sieve uses the norm function $N : \mathbb{Q}(\theta) \to \mathbb{Q}$. 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:

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

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 $N(\alpha) \in \mathbb{Z}$ 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 $N(a + b\theta) \equiv 0 \pmod{p}$ 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. When sufficiently many relations are obtained from the two sieves, they are combined using linear algebra to get the set S of (a, b) pairs such that $\prod_{(a,b)\in S}(a + bm) = y^2$ and $\prod_{(a,b)\in S}(a + b\theta) = \alpha^2$.

3 Implementation Details

In this section, we describe our work on SIMD-based implementations of the sieving step of the MPQSM and that of the NFSM. We assume that the integer to be factored is odd (because powers of 2 can be easily factored out from an even integer). The largest integer factored using a general-purpose algorithm is RSA768 (768 bits, 232 decimal digits). It was factored by Kleinjung et al. on December 12, 2009 [16]. So, we consider integers having up to 250 decimal digits. We have implemented incomplete sieving for the NFSM. Here, we take $d = 3$ if the number of digits in n is less than 80, and $d = 5$ otherwise [3,17]. We have experimented with two integers n_1 (having 60 digits) and n_2 (having 120 digits), where

$$n_1 = 433351150461357208194113824776813009031297329880309298881453$$

is a product of two 30-digit primes, and

$$n_2 = 6337354302474689465336944839067992729056329122332396069074958452797081856763342890897915397385172328002330475832589079 71$$

is a product of two 60-digit primes. Such composite products are frequently used in RSA cryptosystems.

3.1 Sequential Implementation

The sequential implementation is rather straightforward for both the MPQSM and the NFSM.

MPQSM. We take a small prime p from the factor base. Let H be the small integer, up to which there exists a solution of Eqn (4) with $c \in [-M, M]$. For each such $h \in \{1, 2, \ldots, H\}$, we take the precomputed solutions, and for each such solution s, we sieve the array A, that is, subtract $\lfloor 1000 \log p \rfloor$ from the array locations c such that $c = s \pm kp^h$ for some k. We repeat this procedure for all primes in the factor base.

NFSM. We carry out the rational and the algebraic sieves independently on two two-dimensional arrays. The pairs (a, b) indicating smoothness in both the sieves are finally subjected to the test $\gcd(a, b) = 1$. If this gcd is one, then a relation is located. By varying the sieving bound u, we obtain different numbers of relations. We have not attempted to find a complete solvable system. But then, since different cores in one or multiple machine(s) can handle different sieving ranges, and our objective is to measure the benefits of SIMD parallelization, this is not an important issue.

3.2 SIMD-Based Implementation

In our SIMD-based implementations, we have effectively parallelized index calculations. We are provided with 128/256-bit SIMD registers. We want to do one vector addition for 32-bit operands (integers or floating-point numbers) on these registers. For this, two such SIMD registers are loaded with four (or eight) operands each (this is called packing). Then, a single SIMD addition instruction with these two registers as input operands is used to obtain four (or eight) sums in another SIMD register. The four (or eight) results are then extracted from this output register (this is known as unpacking). We carry out this SIMD-based implementation using SSE2 and AVX instruction sets. Notice that unpacking of the output register is necessary for obtaining the four (or eight) indices to subtract log values. However, it is not necessary to repack these indices so long as the sieving bound is not exceeded in these array indices. Unpacking does not destroy the content of an SIMD register, so we can reuse the packed output register as an input during the next parallel index increment.

Implementation on SSE2. While using the SSE2 instruction set, we use 128-bit registers.

MPQSM

- The Basic Idea: This method is similar to the sequential implementation. The major difference is that when we get four integer solutions s_i, we sieve four array locations simultaneously for these four solutions. The four s_i values need not correspond to the same p and/or the same h. However, the s_i values are taken as the minimum solutions in the range $[-M, M]$. We perform four $s_i + kp_i^{h_i}$ operations on 128-bit SIMD registers storing array indices (see Fig. 1), and subtract $\log p$ values from the corresponding array locations.

  A drawback of this method is that four p^h values may vary considerably. For each solution modulo p^h, we sieve at roughly $\lfloor (2M+1)/p^h \rfloor$ locations. If one $p_i^{h_i}$ is larger than other p^h values in a register, the number of iterations of data-parallel sieving is restricted by $\lfloor (2M+1)/p_i^{h_i} \rfloor$, leading to some loss of efficiency. Moreover, if the p^h values in a register are considerably different from one another, the spatial proximity of their sieving locations decreases, and this potentially increases the number of cache misses.

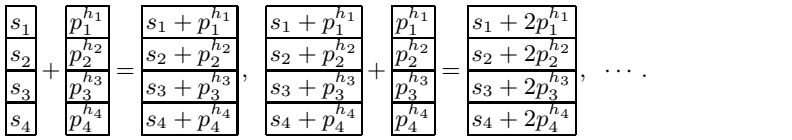

Fig. 1. Parallel index increments during sieving in the MPQSM

– Improvements: The above problems can be reduced to some extent using the following improvement techniques.
  - So far, we have fixed p (a small prime in the factor base) and varied h in the solutions of $T(c) \equiv 0 \pmod{p^h}$. The sieving locations corresponding to different values of h vary considerably. If, on the other hand, we fix h and allow p to vary, the variation in sieving locations is significantly reduced. In other words, we first consume the solutions of $T(c) \equiv 0 \pmod{p_i}$ for $i = 1, 2, 3, \ldots, t$ in groups of four. Next, we process solutions of $T(c) \equiv 0 \pmod{p_i^2}$ for $i = 1, 2, 3, \ldots, t$, again in groups of four, and so on. Now, the quantity $\lfloor (2M+1)/p^h \rfloor$ is roughly of the same order for all of the four p^h values packed in a register. So, the number of iterations in the sieving loop is optimized. Moreover, the probability to hit the same cache line, while accessing locations in A, increases somewhat (particularly, for small values of p and h).
  - If $p^h \geqslant 2M+1$ and $T(c) \equiv 0 \pmod{p^h}$ has a solution $s \in [-M, M]$, then this is the unique solution for c in the range $-M \leqslant c \leqslant M$. We carry out no index calculations $s + kp^h$, but substract $\log p$ only from $A[s]$.
  - If the overhead associated with packing and unpacking dominates over the benefits of parallelization itself, then data parallelization should be avoided. More precisely, for large values of p^h, we have a very few array locations to sieve, and obtaining these sieving locations using SIMD instructions is not advisable to avoid the packing and unpacking overheads. The threshold, up to which parallelizing solutions modulo p^h remains beneficial, depends on M and B, and is determined experimentally.
  - Usually, the congruence $T(c) \equiv 0 \pmod{p^h}$ has two solutions (for an odd prime p). In the rare case where there is a unique solution of this congruence, we perform sieving with respect to this solution sequentially. Indeed, even numbers of solutions are aligned in pairs in 4-segment SIMD registers. An odd number of solutions disturbs this alignment, making the implementation less efficient.

NFSM. Here we do not take into account the exponents h of the small primes in the factor bases. We process four integer solutions for four different p, and we sieve four array locations simultaneously for these four solutions. The initial solutions γ are chosen to be as small as possible in the range $[-u, u]$. We perform four $\gamma + kp$ operations on 128-bit SIMD registers storing array indices, and subtract $\log p$ values from the corresponding array locations. Fig. 2 demonstrates

$$\begin{bmatrix}\gamma_1\\ \gamma_2\\ \gamma_3\\ \gamma_4\end{bmatrix} + \begin{bmatrix}p_1\\ p_2\\ p_3\\ p_4\end{bmatrix} = \begin{bmatrix}\gamma_1+p_1\\ \gamma_2+p_2\\ \gamma_3+p_3\\ \gamma_4+p_4\end{bmatrix}, \quad \begin{bmatrix}\gamma_1+p_1\\ \gamma_2+p_2\\ \gamma_3+p_3\\ \gamma_4+p_4\end{bmatrix} + \begin{bmatrix}p_1\\ p_2\\ p_3\\ p_4\end{bmatrix} = \begin{bmatrix}\gamma_1+2p_1\\ \gamma_2+2p_2\\ \gamma_3+2p_3\\ \gamma_4+2p_4\end{bmatrix}, \quad \cdots .$$

Fig. 2. Parallel index increments during sieving in the NFSM

these parallel index calculations. We again emphasize that unpacking to obtain the individual array indices $\gamma_i + kp_i$ is needed. However, both the packed registers can be reused in all these SIMD additions so long as the primes p_i remain constant, and the array indices $\gamma_i + kp_i$ remain within the sieving bound u.

A similar procedure is followed for the algebraic sieve where solutions of four different (r, p) pairs are taken at a time.

Implementation on AVX. We follow the same basic idea in conjunction with the improvements discussed above. Sandy Bridge comes with 256-bit SIMD registers, using which we can perform vector operations on eight single-precision floating-point data at a time. The AVX instruction set does not support 256-bit vector integer operations. In order to exploit the power of 256-bit registers, we make floating-point index calculations. But then, we also need conversions between floating-point numbers and integers, since array indices must be integers.

Implementation Issues. Some points concerning our parallel implementations (SSE2 and AVX) are listed below.

- To utilize the SIMD registers properly, we break them into 32-bit segments, and vector operations on 32-bit integers and single-precision floating-point numbers are used such that four addition operations take place using a single instruction. We avoid 64-bit vector operations (only two operations at the cost of one instruction) because packing and unpacking overheads can outperform the gain from parallelization. Moreover, in typical factoring algorithms, array indices safely fit in 32-bit unsigned integer values.
- AVX does not provide instructions for 256-bit vector integer operations, so we transform 32-bit integers to 32-bit single-precision floating-point numbers, and conversely. The mantissa segment of 32-bit single-precision floating-point numbers is only 23 bits long (IEEE Floating-Point Standard). This restricts the choice of M, B, u, B_{rat}, and B_{alg}. For example, array indices can be as large as 2^{23} only. However, since this is already a value which is not too small, this restriction is not unreasonable. In fact, we work with these restrictions in our sequential and 128-bit SIMD implementations also. Indeed, only for very large-scale implementations, we need array indices larger than 2^{23}. Even then, this limitation is not a problem so long as each individual core handles a sieving range no larger than 2^{23}.

```
__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. SSE2 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);
```

Fig. 4. AVX intrinsics used for index calculations

- Fig. 3 shows the SSE2 intrinsics used in our implementation.[1] The header file `emmintrin.h` contains the definition of the data type `__m128i` (representing 128-bit registers) and the declarations for the 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 or γ 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 corresponding to `_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 more time-consuming intrinsics `_mm_loadu_si128` and `_mm_storeu_si128`. Another important point is that the packing overhead is high if we attempt to pack from four non-contiguous locations using `_mm_set_epi32` or similar intrinsics. So, we avoid them in our implementations.
- The intrinsics we employ in our implementation using AVX are shown in Fig. 4. The header file `immintrin.h` contains the definition of the data type `__m256` (representing 256-bit registers) and the declarations for the intrinsics `_mm256_load_ps`, `_mm256_add_ps` and `_mm256_store_ps`. 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` and `L`, respectively, using `_mm256_load_ps`. A single vector instruction corresponding to `_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]`,

[1] Only the intrinsics are shown in the figure. The loop structure and other non-SIMD instructions are not shown. The first two intrinsics are used before the sieving loop, whereas the last two intrinsics are used in each iteration of the sieving loop.

`L[2]`, `L[3]`, `L[4]`, `L[5]`, `L[6]` and `L[7]` to integers to obtain the array locations for sieving. If the addresses `P` and `L` are not 32-byte aligned, we need to use the slower intrinsics `_mm256_loadu_ps` and `_mm256_storeu_ps`. We avoid using `_mm256_set_ps` or similar intrinsics which are used to pack eight floating-point numbers from arbitrary non-contiguous locations.

4 Experimental Results and Analysis

4.1 Experimental Setup

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). The GP/PARI calculator (Version 2.5.0) is used to calculate the log values of large integers, to find the zeros of f modulo p (for NFSM), and to validate the results. 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 the GMP library (Version 5.0.5) [18].

4.2 Speeding Up Implementations of the MPQSM Sieve Using SSE2 and AVX

Timing and speedup figures for the implementations of the MPQSM sieve are summarized in Table 1. Timings are reported in milliseconds, and for each n, M, B values we have used in our experiments, we take the average of the times taken by fifty executions. We have incorporated all the improvement possibilities discussed in Section 3.2. 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.

From Table 1, we observe that the speedup is higher for smaller values of B, and increases when the sieving limit M increases. This is expected, since larger sieving bounds or smaller factor base bounds allow parallel index calculations to proceed for a larger number of iterations. On an average, speedup varies between 20–35%, except in the last two clusters where M and B are large. The speedup with AVX is below the expected result and it happens to be almost the same as that with SSE2, despite the use of 256-bit SIMD registers for AVX. The explanation is that, in the AVX implementation, we have to do a lot of conversions between integer and floating-point formats.

4.3 Speeding Up Implementations of the NFSM Sieve Using SSE2 and AVX

Timing and speedup figures for the implementations of the NFSM sieve (in case of n_1 and n_2) are summarized in Table 2 and Table 3, respectively. Timings are measured in milliseconds. For each data set, we record the average of the times

Table 1. Timing and speedup figures for the MPQSM sieve

Number of digits in n	Sieving limit M	Bound on small primes B	Sequential Time (in ms)	SSE2 Parallelization		AVX Parallelization	
				Time (in ms)	Speedup (in %)	Time (in ms)	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 for the NFSM sieve for n_1

	Sieving limit u	Bound on small primes B'	Sequential Time (in ms)	SSE2 Parallelization		AVX Parallelization	
				Time (in ms)	Speedup (in %)	Time (in ms)	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 for 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

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 $1 \leqslant b \leqslant 10$.

From Table 2 and Table 3, we make the following observations. On an average, we get a speedup between 15–40%. The speedup increases with the increase in the sieving limit u. The speedup is found to be somewhat higher for the algebraic sieve (compared to the rational sieve), although we cannot supply a justifiable explanation for this experimental observation.

5 Conclusion

In this work, we have implemented the sieving phase of the MPQSM and that of the NFSM efficiently, using SSE2 and AVX instruction sets. In general, we get a non-negligible performance gain over the sequential (non-SIMD) implementations. This work can be extended in many ways.

- So far, we have implemented only index calculations in a data-parallel fashion. Our efforts on data-parallelizing the subtraction operations have not produced any benefit. More investigation along this direction is called for. Unlike the index-calculation stage, the subtraction stage cannot reuse its output SIMD register.
- In case of the MPQSM, improving the performance of our SIMD-based implementations for large values of M and B deserves further attention.
- In case of the NFSM, the sieving we implemented here is called *line sieving*. A technique known as *lattice sieving* is proposed as an efficient alternative [10]. Data-parallel implementations of lattice sieving are worth studying.
- 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 [19,20]). Fresh experimentation is needed to investigate the effects of SIMD parallelization on polynomial sieves.

References

1. Pomerance, C.: The quadratic sieve factoring algorithm. In: Beth, T., Cot, N., Ingemarsson, I. (eds.) EUROCRYPT 1984. LNCS, vol. 209, pp. 169–182. Springer, Heidelberg (1985)
2. Dixon, B., Lenstra, A.K.: Factoring integers using SIMD sieves. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 28–39. Springer, Heidelberg (1994)
3. 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, pp. 103–126 (1993)
4. Dixon, J.D.: Asymptotically fast factorization of integers. Mathematics of Computation 36, 255–260 (1981)
5. Lehmer, D.H., Powers, R.E.: On factoring large numbers. Bulletin of the American Mathematical Society 37, 770–776 (1931)
6. Morrison, M.A., Brillhart, J.: A method of factoring and the factorization of F_7. Mathematics of Computation 29, 183–205 (1975)
7. Silverman, R.D.: The multiple polynomial quadratic sieve. Mathematics of Computation 48, 329–339 (1987)
8. Lenstra, A.K., Lenstra, H.W., Manasse, M.S., Pollard, J.M.: The number field sieve. In: STOC, pp. 564–572 (1990)
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, pp. 50–94 (1993)
10. Pollard, J.M.: The lattice sieve. In: The Development of the Number Field Sieve. Lecture Notes in Mathematics, vol. 1554, pp. 43–49 (1993)
11. Coppersmith, D., Odlyzko, A.M., Schroeppel, R.: Discrete logarithms in GF(p). Algorithmica 1(1), 1–15 (1986)
12. Lenstra, H.W.: Factoring integers with elliptic curves. Annals of Mathematics 126, 649–673 (1987)
13. Montgomery, P.L.: A block Lanczos algorithm for finding dependencies over GF(2). In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 106–120. Springer, Heidelberg (1995)
14. Microsoft Corporation: Streaming SIMD Extensions 2 Instructions: Microsoft Specific, `http://msdn.microsoft.com/en-us/library/kcwz153av=vs.80.aspx`
15. Intel Corporation: Intrinsics for Intel(R) Advanced Vector Extensions, `http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/intref_cls/common/intref_bk_advectorext.htm`
16. Kleinjung, T., et al.: Factorization of a 768-bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010)
17. Briggs, M.E.: An introduction to the general number field sieve. Master's thesis, Virginia Polytechnic Institute and State University (1998)
18. Free Software Foundation: The GNU Multiple Precision Arithmetic Library, `http://gmplib.org/`
19. Adleman, L.M., Huang, M.D.A.: Function field sieve method for discrete logarithms over finite fields. Information and Computation 151(1-2), 5–16 (1999)
20. Gordon, D.M., McCurley, K.S.: Massively parallel computation of discrete logarithms. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 312–323. Springer, Heidelberg (1993)