CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3211

Straight-line programs for fast sparse matrix-vector multiplication

Samuel Neves*,† and Filipe Araujo

CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal

SUMMARY

Sparse matrix-vector multiplication dominates the performance of many scientific and industrial problems. For example, iterative methods for solving linear systems often rely on the performance of this critical operation. The particular case of binary matrices shows up in several important areas of computing, such as graph theory and cryptography. Unfortunately, irregular memory access patterns cause poor memory throughput, slowing down this operation. To maximize memory throughput, we translate the matrix into a straight-line program that takes advantage of the CPU's instruction cache and hardware prefetchers. The regular loopless pattern of the program reduces cache misses, thus decreasing the latency for most instructions. We focus on the widely used `x86_64` architecture and on binary matrices, to explore several possible tradeoffs regarding memory access policies and code size. We also consider matrices with elements over various mathematical structures, such as floating-point reals and integers modulo m. When compared to a Compressed Row Storage implementation, we obtain significant speedups. Copyright © 2014 John Wiley & Sons, Ltd.

Received 21 March 2013; Revised 31 October 2013; Accepted 23 December 2013

KEY WORDS: large-scale scientific computing; instruction-level parallelism; thread-level parallelism; memory-level parallelism; multi-core architectures and support

1. INTRODUCTION

Linear algebra is at the heart of modern scientific computing. Be it information retrieval, computational physics, graph theory, or cryptography, many problems are often expressible as linear systems, and their solution is often a bottleneck. Thus, the time it takes to solve linear systems has deep repercussions across many areas.

One very common pattern that we find in real-world matrices is *sparsity*, which occurs when most of the matrix coefficients are 0. Sparsity changes the way in which systems can be efficiently solved: typical dense methods, like Gaussian elimination, transform the matrix, adding many nonzeros; this phenomenon is often called fill-in. To avoid fill-in, a different class of methods is usually employed, namely Krylov subspace methods [1]. Krylov subspace methods treat the action of the matrix on a vector (i.e., $b = \mathbf{A}x$) as a *black box*; indeed, these are often called *black box methods*. These methods are particularly suited for sparse matrices, as matrix multiplication can be performed in less than the usual $O(n^2)$ arithmetic operations: an $n \times n$ sparse matrix with ν nonzero entries per row can be multiplied by a vector in $O(n\nu)$. In the real and complex fields, the most common Krylov methods are the CG, Lanczos, and similar subspace correction methods; in finite fields, such as the ones involved in the number field sieve for integer factorization and discrete logarithms [2], the block Lanczos [3] and (block) Wiedemann [4, 5] are the most common.

*Correspondence to: Samuel Neves, CISUC, Department of Informatics Engineering University of Coimbra, Coimbra, Portugal
†E-mail: sneves@dei.uc.pt

In this paper, we study sparse matrices whose coefficients lie in the set $\{0, 1\}$—these are often called 'binary matrices'. This special case is common in spectral graph theory [6] and integer factorization [7–9]. Integer factorization algorithms, such as the number field sieve [2], are able to find factors of large integers after computing solutions to a linear system of the form $\mathbf{A}x = 0$, where $\mathbf{A}$ is a very large binary matrix. For large $n \times n$ matrices, the block Lanczos [3] and block Wiedemann [5], which require $O(n)$ matrix-vector multiplications, are the most popular algorithms. In a recent factorization record [8, 9], the dimension of the matrix solved was $192, 796, 550 \times 192, 795, 550$, with $27, 797, 115, 920$ nonzero entries, totaling about 105 GB in disk space. It was solved using the block Wiedemann method, in roughly 119 days—85% of it spent on sparse matrix-vector multiplications.

Sparse binary matrices also show up in commutative algebra, as part of multivariate equation system solvers. The XL algorithm [10] and its variants solve systems of multivariate equations over some finite field[‡] by considering each appearing monomial as a new variable and finding the solution to a large sparse linear system. Fast linear algebra is often the deciding factor in the success of such algorithms [11, 12], which along with Gröbner basis computation algorithms and SAT solvers are the principal tool to break multivariate equation-based cryptography.

Although we focus on the number field sieve application, where the binary matrix is defined over the $\mathbb{F}_2$ field, that is, the integers modulo 2, it is straightforward to alter the underlying operations to work on the boolean semiring, the integers, or floating-point reals.

The most common bottleneck in sparse matrix-vector multiplication, and thus in linear system solving, is often poor memory access locality, causing suboptimal memory bandwidth [13]. Many different representations try to push memory accesses closer together, to improve locality [14, 15]. One common characteristic of most, if not all, is that they only improve *data* cache accesses, while instruction cache remains largely unused.

In most modern CPUs, however, like Intel's Core 2 and i7 processor lines, as well as in AMD's Phenom and Opteron processors, the instruction cache amounts to half of the total L1 cache. To explore this resource, we propose a machine code representation of binary sparse matrices. Conversion cost from common formats (e.g., Compressed Row Storage) to our format is proportional to the number of nonzero entries of the matrix. In this representation, we can keep the L1 data cache almost exclusively for the input vector, while keeping the matrix itself in the L1 instruction cache; the output vector never resides in cache.

The approach we took in our work was to write a translator to convert the sparse matrix, which represents an $\mathbb{F}_2^n \to \mathbb{F}_2^n$ linear map $y = \mathbf{A}x$, into an `x86_64` straight-line program implementing that very map. This has a number of advantages:

Better L1d cache usage: Regardless of the quality of the representation and ordering used for sparse matrices, they are still using only about $1/2$ of the available L1 cache, the fastest memory available in the processor. Additionally, the matrix's nonzero entries replace entries of the input vector from L1 data cache (L1d), despite the former being used only once.

Flexibility: There is a large volume of published research on reordering, partitioning, and blocking techniques (e.g., [16, 17]). Most of these improvements require changing the representation of the matrix, which also requires new code to use it. Using a straight-line program allows us to freely integrate such techniques without having to change any extra matrix handling functionality. For example, a matrix in the Compressed Row Storage (CRS) format, when converted to, say, Compressed Sparse Block [15] format, will require new code to deal with the new representation. When one adds a new technique to our code representation, only the matrix translator changes, while everything else (i.e., the code that performs the matrix-vector multiplication) remains unchanged (cf. Figure 1).

Implicit prefetching: Modern processors have very elaborate predictive mechanisms to avoid pipeline stalls due to branching mispredictions, false dependencies, and so on [18]. As a result, processors load and decode instructions much before they actually execute them. Our representation is able to take advantage of this quite aggressive instruction prefetching.

[‡]The original XL algorithm was proposed for the $\mathbb{F}_2$ field only; it has been generalized to arbitrary finite fields since.

```
typedef void (*matrix_func_t)(uint32_t *, const uint32_t *);

/* ... */

void spmv(const matrix_func_t matrix, uint32_t *out, const uint32_t *in)
{
    assert(NULL != matrix);
    assert(NULL != in);
    assert(NULL != out);
    matrix(out, in);
}
```

Figure 1. Prototypical C code employed to run a straight-line matrix-vector multiplication program.

We organize the rest of this article as follows. Section 2 describes `x86_64`, the architecture we targeted with our techniques. Section 3 describes our techniques to convert a CRS matrix into code. Section 4 expands the technique to additional mathematical rings beyond the integers modulo 2. Section 5 describes and discusses our experimental results. Section 6 concludes the article, giving some future directions.

2. TARGET ARCHITECTURE

In this article, we target the `x86_64` architecture, as seen in the Intel Sandy Bridge processor line. Nevertheless, we must emphasize that our idea is not restricted to Sandy Bridge. To create machine code for other microarchitectures, such as the Core 2 and AMD's K10, we just need to slightly tune the compiler.

2.1. Pipeline

When executing long runs of non-branching code, there are three things that influence speed: instruction fetch and decoding, instruction dispatch into execution units, and memory bandwidth. The Sandy Bridge pipeline can be split into several main stages: instruction fetching, decoding, dispatching, execution, and retirement.

The instruction fetching in the Sandy Bridge has a total bandwidth of 16 bytes per clock, after which it passes through the predecoder, a mechanism that detects where instructions begin (because of the variable-sized instruction set). The combined throughput of the fetching and (pre)decoding is 16 bytes or four instructions per clock, whichever is smaller [18]. For the best possible fetching throughput[§], the instruction length must average less than 4 bytes.

The instruction decoder in the Sandy Bridge is able to convert up to 32 bytes of code, stored in the decoder queue from the predecoder, into at most 7 μops per cycle. μops are RISC-style instructions actually executed by the execution units in the next stages.

After decoding, instructions are renamed, if needed, and sent to a reservation station. Here, the Sandy Bridge implements dynamic scheduling with six execution units: 3 arithmetic logic units, 2 memory read and address generation units, and 1 memory write unit. Thus, at any given time, it is possible to dispatch at most 6 μops per cycle, two more than the instruction fetcher, the bottleneck, is able to output.

2.2. Memory system

The Sandy Bridge, like most recent CPUs, spends most of its area on caches. It contains two 32KB 8-way set associative L1 caches, and a larger (usually) 256KB 8-way set associative L2 cache for each core. There is also an 12-way associative L3 cache shared between all the cores (cf. Figure 2). Each access to the L1 caches costs 4 cycles, the L2 has a latency of 12 cycles and the L3 about $\approx$ 20 cycles. In each cycle, the L1 caches are able to pull 32 bytes from the L2 caches. The Sandy Bridge also contains an 8-way associative μop cache, which stores up to 1536 *decoded* μops.

[§]This only applies for large streams of code that cannot make use of the μop cache.

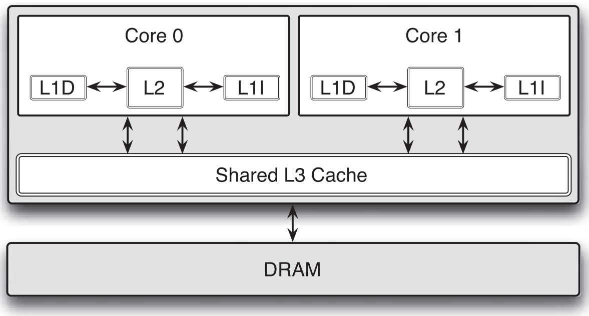

Figure 2. Sandy Bridge memory layout.

Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Figure 3. High-level `x86` instruction encoding.

There are three hardware L1 prefetchers (per core) and two L2 prefetchers in the Sandy Bridge. The three L1 prefetchers are divided into one instruction prefetcher and two data prefetchers. These prefetchers detect access patterns and preload the caches with data likely to be used.

2.3. *Instruction encoding*

In typical CISC fashion, `x86` and `x86_64` have variable-sized instructions. The encoding rules are still mostly backwards compatible with the 1978 8086 processor and have grown to be quite complex [19]. For the scope of this article, however, the reader just needs to know a few basic facts about instruction encoding. Figure 3 describes the basic format of an `x86` instruction, which can range from 1 to 15 bytes.

Prefixes allow instructions to modify the behavior of certain instructions. The `REX.W` prefix (`0x48`), for instance, allows instructions to access the full `x86_64` register set and width; the `0x2E` prefix is a hint to the CPU not to take a branch and so on. There may be multiple prefixes for a single instruction, and some instructions require mandatory prefixes to be accessible.

Opcodes may be 1, 2, or 3 bytes in length. Most common instructions (e.g., integer arithmetic) have 1-byte opcodes, which makes them fairly short.

After the opcodes come the ModR/M and the Scale Index Base (SIB) bytes. These allow an instruction to define source and destination operands flexibly, with one of them possibly being either a register or a memory location. In the latter case, the addressing can be one out of 24 possible modes. The SIB and Displacement bytes are necessary for some of these modes, and the immediate bytes are required for opcodes that require immediates, that is, most arithmetic and move instructions.

For example, an xor of the `EAX` register with the contents of a 32-bit word located at the memory pointed to by `RSI` can be encoded in several different ways:

`33 06`: Simple indirect addressing;
`33 46 00`: Indirect addressing, 8-bit displacement;
`33 86 00 00 00 00`: Indirect addressing, 32-bit displacement.

Assemblers generally choose the shortest possible representation.

3. FAST SPARSE MATRIX-VECTOR MULTIPLICATION

Compressed Row Storage is one of the most common ways to represent (binary) sparse matrices. In this representation, there are three arrays. One array contains the indices for the columns of all nonzero entries in succession. Another one contains the starting positions of each row in the previous array. The last one contains the values of the nonzero entries; in a binary matrix, this array is not required.

```
void spmv(uint32_t *colind, uint32_t *rowidx, uint32_t nrows, uint32_t *in,
    uint32_t *out)
{
    uint32_t i,j;
    uint32_t *ptr = colind;
    for(i=0; i < nrows; ++i)
    {
        const uint32_t ncols = rowidx[i+1] - rowidx[i];
        uint32_t s = 0;
        for(j=0; j < ncols; ++j)
            s ^= in[*colind++];
        out[i] = s;
    }
}
```

Figure 4. A typical Compressed Row Storage (CRS) matrix-vector multiplication implementation in C.

Figure 4 illustrates a simple implementation of sparse matrix-vector multiplication. Because accesses to the `in` vector may be highly irregular, memory latency is often the bottleneck [13]. Many techniques have been proposed over the years, most notably register blocking, cache blocking, software pipelining, software prefetching, translation lookaside buffer caching, among others [14, 20–22].

Our basic idea is to fully unroll a typical CRS implementation, for example, the one from Figure 4 and use the resulting straight-line code sequence as a representation for the matrix itself (when applied to a vector).

Example 3.1
Consider, for example, the following 5×5 matrix:

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

The same linear map represented by $\mathbf{A}$ can alternatively be implemented as

$$\mathbf{A} \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} = \begin{pmatrix} c+e \\ a+c+d \\ e \\ a \\ b+c+d \end{pmatrix}.$$

One could simply convert the mapping into, say, C code, and let an optimizing compiler choose the best combination of instructions and memory access patterns. As matrices grow, however, compilers are unable to generate that volume of code in reasonable time (cf. [23]). As such, we must do the compilation and optimization of the matrix ourselves. This section describes the various approaches and improvements possible on the target (`x86_64`) architecture.

3.1. Approach A: baseline

Our initial approach converts this mapping to `x86_64` code in the most straightforward way possible: convert the arithmetic operations of a regular CRS matrix-vector multiplication (cf. Figure 4) to a straight-line function, callable from a compiled language such as C or FORTRAN. We use 32-bit registers whenever possible, to avoid the `REX.W` prefix of full 64-bit operations, which would cost 1 extra byte per instruction. Additionally, we use the `STOSD` instruction, only 1 byte long, to store the result of each row into the output vector.

```
; void spmv(u32 *out, u32 *in)
; RSI contains source, RDI destination
; Row 1
mov eax, [rsi +  8] ; 8B 46 08
xor eax, [rsi + 16] ; 33 46 10
stosd               ; AB
; Row 2
mov eax, [rsi]      ; 8B 06
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
stosd               ; AB
; Row 3
mov eax, [rsi + 16] ; 8B 46 10
stosd               ; AB
; Row 4
mov eax, [rsi]      ; 8B 06
stosd               ; AB
; Row 5
mov eax, [rsi +  4] ; 8B 46 04
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
stosd               ; AB
ret                 ; C3
```

Figure 5. Straight-line program implementing the linear mapping represented by matrix **A**.

For an $n \times n$ matrix with ω nonzero entries, this approach requires at most $n + 6\omega$ bytes of storage; the standard CRS format with 32-bit indices requires $4n + 4\omega$ bytes. Figure 5 shows the `x86_64` assembly code[¶] of our initial approach.

3.2. Approach B: reducing code size

One observation we made was that there are three different instruction sizes for different displacements. When there is no displacement (e.g., `MOV EAX`, [RSI]), the instruction requires 2 bytes. When the displacement is between -128 and 127 (e.g., `MOV EAX, [RSI+32]`), the instruction requires 3 bytes. Finally, for 32-bit displacements (e.g., `MOV EAX, [RSI+12345678]`), the instruction requires 6 bytes. For example, in Figure 5, all but one access need 3 bytes, because the associated input and output vectors fit perfectly within 256 bytes.

Our first improvement is to take advantage of the smaller encoding for $[-128; 127]$ displacements and try to create as many of them as possible. We do this by moving the `RSI` register forward whenever there is a 'burst' of nonzero entries close together. Moving `RSI` costs 6 bytes—the savings are 3 bytes per entry. Thus, we only advance `RSI` when there are at least three nonzero entries in a 256-byte interval. Because we are changing `RSI`, we must reset to its original value in the beginning of each row: we do this using the 1-byte `PUSH` and `POP` instructions, to store and recover `RSI`, respectively.

In the worst case, this improvement does not improve anything at all, leaving the code size at $n + 6\omega$ bytes. In the best case, where every nonzero entry is in a 256-byte neighborhood, we obtain $3n + 3\omega + \frac{6}{64}\omega$ bytes.

3.3. Approach C: register blocking

Register blocking is a technique already known from sparse linear algebra optimization [14] that we employ quite literally. Instead of using simply one register and traversing the matrix row by row, we use a number of registers B and traverse the matrix B rows at a time. Whenever two elements of vector x are accessed by two or more rows in the same register block, only one memory load is issued. This saves memory bandwidth, decreases code size, and increases instruction-level parallelism. Plus, unlike register blocking in usual data formats (e.g., [14]), code register blocking does not add fill overhead due to storing extra 0s to make the blocks dense.

[¶]Our converter outputs actual machine code, not assembly mnemonics; we show instead the commented assembly code for readability purposes.

```
; First nonzero column
mov eax, [rsi +  8] ; 8B 46 08
mov ebx, [rsi]      ; 8B 1E
mov ecx, [rsi + 16] ; 8B 4E 10
mov edx, ebx        ; 89 DA
; Second nonzero column
xor eax, [rsi + 16] ; 33 46 10
xor ebx, [rsi +  8] ; 33 5E 08
; Third nonzero column
xor ebx, [rsi + 12] ; 33 5E 0C
; Store block
stosd               ; AB
xchg eax, ebx       ; 93
stosd               ; AB
xchg eax, ecx       ; 91
stosd               ; AB
xchg eax, edx       ; 92
stosd               ; AB
; Row 5 --- part of a new block
mov eax, [rsi +  4] ; 8B 46 04
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
stosd               ; AB
retn                ; C3
```

Figure 6. Straight-line program for **A** using basic register blocking ($B = 4$).

We have implemented two variants of register blocking: one with block size 4 and another with block size 12. The former only uses the registers available in the IA-32 instruction set (EAX, EBX, ECX, and EDX—ESI and EDI are used for addressing, ESP for stack and EBP as a temporary variable), guaranteeing that no REX prefix bytes are used. The latter uses all available general-purpose registers in the x86_64 architecture (RAX–R15 except {RSI,RDI,RBP}), minus four registers used for the vector addresses, stack, and a temporary variable. Figure 6 shows the assembly code for matrix **A** with register blocking ($B = 4$).

To store row results, we still prefer to use the STOSD instruction. Because STOSD has the implicit argument EAX for the value to save, we use the XCHG instruction, only 1 byte long, to load values onto EAX, and STOSD to save them to the output vector (cf. Figure 6). Using XCHG coupled with STOSD seems to be the shortest possible method to store the output vector elements, with every other approach being at least 3 bytes long.

In the worst case, the $B = 4$ variant (applied to the base method of Section 3.1) uses $2n + 6\omega$ bytes, and the $B = 12$ variant uses $\frac{2\cdot4+3\cdot9}{13}n + \frac{4\cdot6+9\cdot7}{13}\omega$ bytes. In the best case, that is, when every row has the same entries, we obtain $4n + 3n + 2\omega$ and $4n + 3n + \frac{2\cdot4+3\cdot9}{13}\omega$ bytes, for $B = 4$ and $B = 12$, respectively.

3.4. Approach D: sorted register blocking

From the example in Figure 6, it is easy to see that the basic register blocking method of Section 3.3 is not performing optimally. There are needless duplicate loads of both [RSI + 16] and [RSI + 8], because their column indexes do not coincide.

Instead of blindly converting the matrix operations as they appear on its CRS representation, one can instead sort the order of operations by memory accesses. Because the CRS rows are sorted to begin with, this is a very fast merge operation. This has no negative influence on code size, as it is simply a reordering of instructions, but has several speed advantages: memory accesses become optimal (up to block size), and there is an increased likelihood of entries within the same block being less than 256 bytes apart. Figure 7 shows the assembly listing for the sorted register blocking output of matrix **A**.

3.5. Approach E: uncached stores

Up until now, we have been employing the STOSD instruction to perform the memory stores of the result of each row's inner product with the input vector. STOSD, however, brings the whole cache line corresponding to that memory location to cache and modifies it, possibly evicting input vector values from cache.

```
mov ebx, [rsi]      ; 8B 1E
mov edx, ebx        ; 89 DA
mov eax, [rsi +  8] ; 8B 46 08
xor ebx, eax        ; 31 C3
xor ebx, [rsi + 12] ; 33 5E 0C
mov ecx, [rsi + 16] ; 8B 4E 10
xor eax, ecx        ; 31 C8
; Store
stosd               ; AB
xchg eax, ebx       ; 93
stosd               ; AB
xchg eax, ecx       ; 91
stosd               ; AB
xchg eax, edx       ; 92
stosd               ; AB
; Row 5 --- part of new block
mov eax, [rsi +  4] ; 8B 46 04
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
stosd               ; AB
retn                ; C3
```

Figure 7. Straight-line program for **A** using sorted register blocking ($B = 4$).

```
; Row 1
mov eax, [rsi +  8] ; 8B 46 08
xor eax, [rsi + 16] ; 33 46 10
movnti [edi], eax   ; 0F C3 07
; Row 2
mov eax, [rsi]      ; 8B 06
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
movnti [edi +  4], eax ; 0F C3 47 04
; Row 3
mov eax, [rsi + 16] ; 8B 46 10
movnti   [edi +  8], eax ; 0F C3 47 08
; Row 4
mov eax, [rsi]      ; 8B 06
movnti [edi +  12], eax ; 0F C3 47 0C
; Row 5
mov eax, [rsi +  4] ; 8B 46 04
xor eax, [rsi +  8] ; 33 46 08
xor eax, [rsi + 12] ; 33 46 0C
movnti  [edi +  16], eax ; 0F C3 47 10
retn                ; C3
```

Figure 8. Straight-line program implementing the linear mapping represented by matrix **A**, with nontemporal stores.

To avoid cache pollution, we employ the SSE2 instruction MOVNTI. This instruction is one of several *nontemporal* store operations provided by the SSE2 instruction set. Nontemporal instructions give a hint to the processor that the stored data are not going to be used anytime soon and therefore do not need to be brought to cache memory. When applied to the base code from Section 3.1, this tweak slightly increases the code size to at most $4n + \frac{6}{64}n + 6\omega$ bytes. Figure 8 displays the code for **A** when using MOVNTI.

3.6. *Approach F: extended register blocking*

In Section 3.3, we have introduced register blocking to improve memory locality and software parallelism of the multiplication, limiting ourselves to general-purpose registers. This need not be so: using the SSE4.1 instruction set, one can use the PINSRD and PEXTRQ instructions to effectively turn the 16 XMM registers into an addressable fast memory space to store a register block. This increases the register block size by 64, to a maximum of 76 possible registers. Additionally, using the MMX registers increases the potential block size to 92. Figure 9 illustrates the workings of extended register blocking.

In Sandy Bridge processors, one can take register blocking even further. The AVX instruction set extension introduces the 256-bit YMM registers, which are in reality an extension of XMM registers. In fact, XMM registers in AVX are simply an alias to the lower 128 bits of YMM registers. This allows us to duplicate the amount of registers, although it is less straightforward than in XMM registers: YMM

XMM0	b_3	b_2	b_1	b_0
XMM1	b_7	b_6	b_5	b_4
XMM2	b_{11}	b_{10}	b_9	b_8
XMM3	b_{15}	b_{14}	b_{13}	b_{12}
XMM4	b_{19}	b_{18}	b_{17}	b_{16}
XMM5	b_{23}	b_{22}	b_{21}	b_{20}
XMM6	b_{27}	b_{26}	b_{25}	b_{24}
XMM7	b_{31}	b_{30}	b_{29}	b_{28}
XMM8	b_{35}	b_{34}	b_{33}	b_{32}
XMM9	b_{39}	b_{38}	b_{37}	b_{36}
XMM10	b_{43}	b_{42}	b_{41}	b_{40}
XMM11	b_{47}	b_{46}	b_{45}	b_{44}
XMM12	b_{51}	b_{50}	b_{49}	b_{48}
XMM13	b_{55}	b_{54}	b_{53}	b_{52}
XMM14	b_{59}	b_{58}	b_{57}	b_{56}
XMM15	b_{63}	b_{62}	b_{61}	b_{60}

```
pextrd eax, xmm6, 1 ; Access reg. 26
pinsrd xmm6, eax, 1 ; Insert back
```

Figure 9. How to use the XMM register set as an array of addressable 32-bit integers.

registers do not permit easy extraction of words from their *upper* half. To get around this limitation, we can use the VPERM2F128 instruction to swap the lower and upper halves of a YMM register, thus enabling all words to be accessible via the PEXTRD and PINSRD instructions. This requires some bookkeeping from the translator side.

This method has, however, a big disadvantage: code size. While (general-purpose) register blocking has no overhead and often even saves space, extended register blocking requires 12 bytes per register block access, which is unacceptable. Applying this technique to the base case of Section 3.1 would put the best case scenario of the code size at $n + 18\omega$ and the worst case at $n + 21\omega$; the YMM trick may increase this figure to up to $n + 27\omega$. Thus, we do not use this method for large matrices. In comparison, using the stack memory to store temporary elements would cost $n + 12\omega$ bytes.

3.7. Approach G: memory prefetching

When one converts a sparse matrix into code, it becomes essentially unnecessary to try and prefetch the matrix itself, as the processor's hardware prefetcher (cf. Section 2) already takes care of code prefetching. There is, however, still room for software prefetching when it comes to the input vector. In particular, because we have exact knowledge of all memory accesses, we are able to prefetch values that are far away and to prefetch the first value of a new row before the current one is finished. The prefetching instructions (PREFETCHT0 for Intel and PREFETCH for AMD) are not short; however, each useful prefetch instruction costs 7 bytes.

For such a prefetch instruction to be useful, it must necessarily avoid, or decrease the chance, of an input vector cache miss. A simple policy, often used in CRS and CRS-variants, is to prefetch input vector elements and indices that are to be used in future rows; there is no need to prefetch indices in our case, as they're embedded in code. Placing a prefetch instruction on every addition operation, as is typical in many implementations (e.g., [24]), is clearly not practical: 7 extra bytes per operation results in a matrix with possibly more than double the size, which would displace more bandwidth from vector accesses to code fetching. We opted not to insert prefetch instructions. With that said, depending on the target matrix structure, with some additional analysis of matrix memory accesses, and coupled with a cache model approximating that of the host CPU, it may be possible to insert a relatively small amount of prefetch instructions that have a significant impact in overall performance.

3.8. Approach H: combinatorial optimization

Although algebraic methods to reduce operation count in (dense) binary matrices are an open problem and likely impossible [25], combinatorial approaches are known and are quite common [26, 27]. This raises the question: can we use such combinatorial methods to improve arithmetic operation and memory access counts in sparse matrices?

In Section 3.3, we have already performed a simple form of combinatorial optimization, by omitting the loading of repeated column entries within the same block. One could continue through this route and search for common subexpressions that would reduce the operation count and code size. The problem with this approach is threefold:

Combinatorial algorithms are slow: All combinatorial algorithms currently in use require at least $n^{2+o(1)}$ operations to improve operation counts by at most a $(\log n)^2$ factor [25–27]. In sparse matrices, however, $n^{2+o(1)}$ is the expected complexity to solve a full linear system!

Most combinatorial algorithms require temporary space: Most combinatorial algorithms, with the exception of the fully in place method of Bernstein [27][‖], require storing many temporary variables during runtime. Input partitioning[**], for example, requires the computation of $n^2/\log^2 n$ temporary sums; even adjusting for the sparsity of the target matrix, this is a prohibitive amount of space.

Combinatorial algorithms exploit redundancy: Combinatorial 'non-algebraic' approaches to matrix-vector multiplication exploit certain redundancies when matrices are understood as graphs. In sparse matrices, redundancies are already partially eliminated by the representation: only the useful nonzeros are stored. Further redundancies are few and far between, as there are simply less possible combinations to explore.

Combinatorial methods may nevertheless be useful for matrices that have small dense subsets, by minimizing the arithmetic operation count in those areas. In general, however, there seems to be little benefit of such techniques on large sparse matrices.

3.9. Parallelization

Working on scientific problems, it is expected that we need to deal with very large matrices; this exacerbates the need for parallel and distributed algorithms. Typical methods to parallelize sparse matrix-vector multiplication divide the matrix into blocks, and process each block individually, with minimal synchronization between them. There are many known heuristics to choose blocks in sparse matrices, so as to minimize the total runtime [7]. These heuristics are compiler-agnostic: one can divide the matrix in any desired way, before converting it to `x86_64` code.

Synchronization between threads, in any of the partitioning schemes, can be improved by using atomic instructions. In the case of the integers, $\mathbb{F}_2$, and the boolean semiring, one can use the `LOCK` prefix to perform atomic additions to destination vectors, thus avoiding explicit locks and their respective issues. For example, {add,xor,or}-ing a 32-bit register to a memory location (say, the output vector) can be performed simply with the 3-byte instruction `LOCK {ADD,XOR,OR} [EDI], EAX`. Note that, while the same instructions without the `LOCK` prefix would have a $\approx$ 6-cycle latency[††] on Sandy Bridge, the latency grows to about 25 cycles when the `LOCK` prefix is added. Furthermore, the `LOCK` prefix does prevent other cores from accessing memory during its execution. Thus, although hardware support for atomic operations is an asset, the usual contention considerations must be taken into account to achieve good performance.

3.10. Summary

Table I summarizes the code sizes of the techniques we proposed in this section. To recap, n is the number of rows in the matrix, and ω is its weight. All sizes after Approach A from Section 3.1 refer to each technique implemented individually on top of this base case and not cumulatively.

[‖]The [27] method, however, destroys the input vector; this can be seen as a special kind of temporary storage.

[**]Also known as 'Method of Four Russians', following the names of its original authors [26]; it also has been attributed to earlier work of Lupanov [28].

[††]This assumes the memory address in question already is loaded into L1 cache.

Table I. Code sizes of the various approaches described in Section 3.

Approach	Best case	Worst case
CRS	$4n + 4\omega$	$4n + 4\omega$
A	$n + 6\omega$	$n + 6\omega$
B	$3n + 3.09\omega$	$5n + 6\omega$
C ($B = 4$)	$4n + 2n + 2\omega$	$2n + 6\omega$
C ($B = 12$)	$4n + 2.69n + 2\omega$	$2.69n + 6.69\omega$
D ($B = 4$)	$4n + 2n + 2\omega$	$2n + 6\omega$
D ($B = 12$)	$4n + 2.69n + 2\omega$	$2.69n + 6.69\omega$
E	$4.09n + 6\omega$	$4.09n + 6\omega$
F	$n + 18\omega$	$n + 21\omega$

4. MATRICES IN OTHER RINGS

Up to this point, we focused on the ring of integers modulo 2, where addition is performed with bitwise xor, and multiplication with bitwise and. This section describes some other rings where one may also transform sparse matrix-vector multiplication into code.

4.1. *Boolean semiring*

The boolean semiring, where addition is a bitwise or, and multiplication is a bitwise and, is extremely similar to the modulo 2 case, and every aspect of Section 3 applies here. The only difference is that one uses the `or` instruction in place of the `xor`; every size and performance aspect remains unchanged (cf. Figure 10).

4.2. *Floating-point reals*

Floating-point is possibly the most common ring associated with matrix operations. Maintaining the assumption of a binary matrix, we can use the x87 `FLD`, `FADD`, and `FST` instructions in place of the `MOV`, `XOR`, and `STOSD/MOVNTI` instructions, as exemplified in Figure 11(a). Note that although we focus on single-precision floating-point instructions here, every aspect carries over to double precision trivially.

The code size of the x87 variant is similar to its modulo 2 counterpart but has a significant drawback: because of the stack-based nature of the x87 floating-point registers, it is required to use an additional 2-byte instruction, `FXCH`, to be able to treat the 8 FPU registers as a register file (as opposed to a stack). This limits the effectiveness of register blocking. Furthermore, because of the lack of nontemporal store instructions on these registers, stores are always cached.

The alternative is to use the `XMM` register set and its associated instructions: `MOVSS`, `ADDSS`, and so on. The drawback is that such instructions are longer, requiring 2 extra bytes per instruction

```
mov eax, [rsi +  8] ; 8B 46 08
or  eax, [rsi + 16] ; 0B 46 10
stosd               ; AB
mov eax, [rsi]      ; 8B 06
or  eax, [rsi +  8] ; 0B 46 08
or  eax, [rsi + 12] ; 0B 46 0C
stosd               ; AB
mov eax, [rsi + 16] ; 8B 46 10
stosd               ; AB
mov eax, [rsi]      ; 8B 06
stosd               ; AB
mov eax, [rsi +  4] ; 8B 46 04
or  eax, [rsi +  8] ; 0B 46 08
or  eax, [rsi + 12] ; 0B 46 0C
stosd               ; AB
ret                 ; C3
```

Figure 10. Straight-line program implementing the linear mapping represented by matrix **A** over the boolean semiring.

```
fld  dword [rsi +  8] ; D9 46 08
fadd dword [rsi + 16] ; D8 46 10
fst  dword [rdi]      ; D9 17
fld  dword [rsi]      ; D9 06
fadd dword [rsi +  8] ; D8 46 08
fadd dword [rsi + 12] ; D8 46 0C
fst  dword [rdi + 4]  ; D9 57 04
fld  dword [rsi + 16] ; D9 46 10
fst  dword [rdi + 8]  ; D9 57 08
fld  dword [rsi]      ; D9 06
fst  dword [rdi + 12] ; D9 57 0C
fld  dword [rsi +  4] ; D9 46 04
fadd dword [rsi +  8] ; D8 46 08
fadd dword [rsi + 12] ; D8 46 0C
fst  dword [rdi + 16] ; D9 57 10
ret                   ; C3
```

(a) x87 instruction set.

```
movss xmm0, [rsi +  8] ; F3 0F 10 46 08
addss xmm0, [rsi + 16] ; F3 0F 58 46 10
movss [rdi], xmm0      ; F3 0F 11 07
movss xmm0, [rsi]      ; F3 0F 10 06
addss xmm0, [rsi +  8] ; F3 0F 58 46 08
addss xmm0, [rsi + 12] ; F3 0F 58 46 0C
movss [rdi + 4], xmm0  ; F3 0F 11 47 04
movss xmm0, [rsi + 16] ; F3 0F 10 46 10
movss [rdi + 8], xmm0  ; F3 0F 11 47 08
movss xmm0, [rsi]      ; F3 0F 10 06
movss [rdi + 12], xmm0 ; F3 0F 11 47 0C
movss xmm0, [rsi +  4] ; F3 0F 10 46 04
addss xmm0, [rsi +  8] ; F3 0F 58 46 08
addss xmm0, [rsi + 12] ; F3 0F 58 46 0C
movss [rdi + 16], xmm0 ; F3 0F 11 47 10
ret                    ; C3
```

(b) SSE instruction set

Figure 11. Straight-line program implementing the linear mapping represented by matrix **A** over single-precision floating-point: (a)x87 instruction set and (b)SSE instruction set.

```
_label:
sub ecx, ebp ; 29 E9
jl _label    ; 72 FC
```

(a) Using a loop.

```
xchg ecx, eax ; 91
cdq           ; 99
div  ebp      ; F7 F5
mov  ecx, edx ; 89 D1
```

(b) Using a DIV.

```
mov  eax, ecx  ; 89 C8
mul  ebp       ; F7 E5
add  rdx, rcx  ; 48 01 CA
shr  rdx, 7    ; 48 C1 EA 07
imul edx, 113  ; 6B D2 71
sub  ecx, edx  ; 29 D1
```

(c) Using multiplication.

Figure 12. Sample reduction of ECX modulo 113 using three alternative techniques: (a) using a loop; (b) using a DIV; and (c) using multiplication.

as mandatory prefixes (cf. Figure 11(b). Using XMM registers seems, however, preferable, because x87 requires the use of the 2-byte FXCH instruction for all but the most basic of code generation approaches.

Note that in the floating-point case, we still assume that the coefficients are in the $\{-1, 0, 1\}$ set. It is not practical to allow arbitrary floating-point coefficients, because the instruction set does not allow immediate operands, and therefore, the coefficients would have to be stored in a similar fashion as in CRS.

4.3. *Integers modulo m*

Matrices with coefficients in the ring of integers modulo some small prime, resulting from discrete logarithm computations or multivariate equation system solvers such as XL are also fairly common. Replacing the XOR instruction by ADD changes the ring to the integers modulo 2^{32}, while preserving the size of the code. Reducing modulo any smaller power of 2 is trivially achieved by adding an AND instruction at the end of each row.

However, when the modulus is not a power of 2, reducing the sums in each row is trickier. What is the best way to reduce an integer modulo m? There are three options[‡‡] (cf. Figure 12):

- Use a loop: subtract the modulus from the partial sum until it is smaller than the modulus;
- Use the DIV instruction;
- Use multiplication and shifting to perform division, as described by Granlund and Montgomery [29].

Neither of these options is very attractive: looping can waste many cycles if the sum is much larger than the modulus, DIV is a very slow and non-pipelined instruction, and clobbers RAX and RDX, and dividing using multiplication requires a fairly large sequence of instructions, which involves tens of bytes and also clobbers RAX and RDX.

[‡‡]For simplicity, we assume that non-power-of-2 moduli are odd and prime.

The choice of what to do depends on the relative size of the modulus compared to the register size. For example, if the modulus is $m = 113$ (7 bits), it is possible to perform around 2^{25} additions without having to reduce modulo m; in this case, we may prefer `DIV` or Granlund's method once at the end of each row. On the other hand, if the modulus is close to the register size (e.g. 28 bits), it may be best to reduce the partial sum every $2^{32-\lceil \log_2 m \rceil} - 1$ additions using a loop. Alternatively, one can instead *increase* the register size to 64 bits (at the cost of 1 extra byte per operation) and reduce once per row as described earlier.

Let ν be the average number of elements per row, and $\lambda = 2^{32-\lceil \log_2 m \rceil} - 1$ be the maximum number of additions one can safely perform before an overflow. The overhead of modular reduction per row for each of the three above methods is bounded by the following:

- 5 bytes when m is a power of 2;
- $\lceil 4\nu/\lambda \rceil$ bytes for the loop;
- $\lceil 6\nu/\lambda \rceil$ bytes for `DIV`;
- $\lceil 16\nu/\lambda \rceil$ bytes for division using multiplication, when $m < 128$ and registers are 32 bits;
- $\lceil 19\nu/\lambda \rceil$ bytes for division using multiplication, when $m \geqslant 128$ and registers are 32 bits.

Note that while the division by multiplication approach does have a large size footprint, it is the fastest approach, and it is recommended for cases when reduction is only performed very few times. The critical factor here is ν/λ, which acts as an overhead multiplier.

For example, a matrix where $\nu = 60$ and $m = 113$ has an effective overhead of $\lceil 60/(2^{32-7} - 1) \rceil = 1$, in which case an overhead of 16 bytes per row is acceptable. On the other hand, if $m = 2^{31} - 1$, we have $\lceil 60/(2^{32-31} - 1) \rceil = 60$ extra reductions per row, which may end up dominating code size. In this case, it is preferable to minimize overhead, by either using the loop approach, or by increasing register size to 64 bits. The latter option increases every arithmetic operation by 1 byte, because of the `REX` prefix, but it is still shorter than performing all the necessary modular reductions on a 32-bit register.

5. RESULTS AND DISCUSSION

To benchmark our code, we used a matrix obtained from a 512-bit number field sieve factorization job. This matrix, `large.crs`, is $5,426,753 \times 5,426,928$ in size, with $370,909,586$ nonzero entries. Because of size concerns, we did not test the extended register blocking of Section 3.6, because we did not have enough memory. The test machine contained an Intel Core i7 2630QM CPU, using DDR3 RAM clocked at 1333 MHz.

We tested our CRS implementation in $\mathbb{F}_2$ (based on [24]) against five combinations of the techniques of Section 3:

Combination 1: Approach A
Combination 2: Approach A + B
Combination 3: Approach A + B + D ($B = 4$)
Combination 4: Approach A + B + D ($B = 12$)
Combination 5: Approach A + B + D ($B = 12$) + E

Our first test concerns our key performance figures: time (measured in clock cycles) and code size. Code size is measured in bytes per nonzero matrix entry.

Table II lists those performance figures when measured for the `large.crs` matrix, roughly 1.5 GB in size when stored in CRS format. This matrix puts much pressure into every part of the memory subsystem, and our simpler implementations fall very close to the CRS implementation. As our methods improve, so does the performance of our code: register blocking reveals to be a real asset, even when it increases the code size. It is also worth to point out that, while the register blocked code is indeed slightly larger, it is made up of much smaller instructions than the naïve (cf. Figure 7)—in average, the register blocked code (with `RSI` updates) generates 2 and 3 byte

Table II. Speeds (in clock cycles) and bytes per nonzero entry for the `large.crs` matrix.

Method	CRS	1	2	3	4	5
Cycles / 1000	8,583,385	7,453,025	5,476,997	5,107,850	4,794,578	4,681,212
Bytes/nonzero	4.0245	6.0146	5.88905	5.63507	6.09096	6.13851

Table III. Comparison of cache misses between the CRS implementation and Combination 5.

Method	L1d	L1i	L2	L3
CRS	378,853,972	12,038	373,698,765	283,165,339
Combination 5	306,808,927	35,678,176	326,558,944	246,930,244

CRS, Compressed Row Storage.

Table IV. Speedup (over) and size increase (under) observed with Combination 5 against CRS, for varying dimension n and density d.

	d					
n	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
10^3	2.74	3.85	4.11	—	—	—
	0.99	1.26	1.71	—	—	—
10^4	2.02	2.22	2.79	3.97	—	—
	0.98	1.14	1.73	1.91	—	—
10^5	—	3.26	2.97	2.25	3.52	3.51
	—	1.13	1.64	1.78	1.88	1.89
10^6	—	—	—	2.73	1.31	1.35
	—	—	—	1.68	1.77	1.85

CRS, Compressed Row Storage.

instructions, the optimal size on the target architecture (cf Section 2.1). Further, it also provided much more instruction-level parallelism, by running several independent rows, allowing the execution units to work while waiting for memory loads. Our best implementation runs in about 54% less cycles than CRS, despite being 52% larger.

Another point of interest to us was the behavior of the cache memory. To measure it, we employed the hardware counters available in most common processors and made easily available to us by the Performance Application Programming Interface library [30]. Table III shows the cache behavior of our best-behaving implementation and of the CRS stored implementation. The first thing we noticed is how the CRS has almost no instruction cache misses. This is natural, because even an optimized CRS implementation will generally have a small enough loop to fit within the processors's loopback buffer [18]. Our implementation has a relatively high number of instruction cache misses, which is to be expected because of the large code size. Its other (L1d, L2, and L3) cache misses, however, are all lower than CRS; the total sum of misses of our method saves roughly 120 million cache misses overall, which results in its higher performance.

We have performed additional experiments, now with randomly generated matrices of various dimensions and density. Table IV shows the speedup and the size increase, in relation to CRS, of Combination 5.

The results obtained in Table IV are coherent with the previous measurements, and show that the real-world matrix originally employed—with very large dimension and extremely high sparsity—is actually the worst-case scenario for our method.

We have also performed experiments where we partitioned the matrix and ran it in all four cores of our processor and measured the observed speedup. To minimize noise in the measurements, both the hyper-threading and turbo boost CPU features were disabled. The results are in Table V. The

Table V. Speedup observed with Combination 5 against itself when using four threads, for varying matrix dimensions (n) and densities (d).

	d					
n	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}
10^3	3.59	2.26	1.21	—	—	—
10^4	2.16	3.32	3.93	3.69	—	—
10^5	—	1.99	2.32	2.36	2.83	3.53
10^6	—	—	—	3.23	3.44	3.30

Table VI. Matrix translation and execution timings for the `small.crs` matrix.

Method	CRS	1	2	3	4	5
Translation time	—	0.461	0.450	0.920	0.950	0.995
Execution time	0.028	0.030	0.029	0.026	0.024	0.024

Table VII. Matrix translation and execution timings for the `large.crs` matrix.

Method	CRS	1	2	3	4	5
Translation time	—	28.75	28.59	44.19	51.35	51.69
Execution time	4.291	3.726	2.738	2.553	2.397	2.340

partitioning scheme was naïve row partitioning—we are not benchmarking partitioning schemes here—so there is room for improvement. We also did not employ the `LOCK`-based synchronization described in Section 3.9 but used simple locking primitives instead. That said, the results, particularly for large matrices, are encouraging and suggest parallelization of our straight-line code deserves further study.

Finally, it is important that the translation from CRS to straight-line program be reasonably fast—if the translation takes longer than solving the linear system, it is hardly worth it—therefore, we also measured the time required to translate the matrix into `x86_64` code. For this test, we use both the previous `large.crs` matrix, and a smaller NFS-generated matrix, `small.crs`, which is $150,615 \times 150,802$ and has weight $14,599,768$. The results are shown in Tables VI and VII. While the translation time is higher than the execution time, it is worthwhile to preprocess the matrix for, say, an iterative solver, where the expected number of matrix-vector multiplications is proportional to the number of rows [§§].

For example, using Method 5 to translate the matrix is roughly 35 times slower than a single CRS matrix multiplication, as seen in Table VI. However, on a matrix of large dimension N, this is an acceptable cost: the total wall time will be $35 \cdot t_M + N \cdot \frac{t_M}{s}$, where t_M is the time to multiply a matrix in CRS, and s is the speedup obtained from the translation. For large enough matrices, such as in our examples, this clearly results in a net speedup. The same observations apply to `large.crs`, where the disparity between translation time and CRS time favors our approach even more.

It is also worth noting that our translation method scales essentially linearly with the matrix weight, because it processes the matrix in similar fashion as a standard CRS multiplication. We can see from Table VII that the `large.crs` matrix, while around 25 times heavier than `small.crs`, has a translation time roughly 50 times higher. The factor of approximately 2 between perfect linear scaling and reality is mainly attributable to the larger matrix size, which causes worse memory performance (i.e., cache misses and page faults) during translation.

[§§]This is especially true for matrices coming from factorization and discrete logarithms, over some finite field. In real, floating-point, matrices the average number of iterations can be much smaller, often in the range of tens or hundreds.

6. CONCLUSION

In this article, we have studied the implications and consequences of fully converting a binary sparse matrix to code. Our method takes advantage of the L1 instruction cache, and sorted register blocking allied with nontemporal hints and careful size-coding allowed us to produce straight-line programs representing the same linear map in relatively small space, and up to 4 times higher performance.

While most of our focus was on the arithmetic of $\mathbb{F}_2$ matrices for integer factorization, our ideas and methods have broader applications, such as graph mining. Some of those matrices, although in principle belonging to the real numbers, only have small entries, say, in the set {0,1,-1}. One can use our methods to convert such matrices to code, replacing the `XOR` instruction by `FADD/FSUB` or `ADDPS/SUBPS` or `ADDPD/SUBPD`, and the `MOV` instruction by `FLD/FST` or `MOVSS` or `MOVSD`. Matrices over the integers modulo m are also possible and feasible, by performing modular reductions only when strictly necessary.

Fast linear algebra, despite decades of intense research, is once again a flourishing area, with the advent of (massively) multi-core processors. We have performed some preliminary work on parallelization, showing that there is still ground to cover to fully take advantage of the processor and memory hierarchy in the parallel setting.

ACKNOWLEDGEMENTS

This work has been partially supported by the project PTDC/EIA-EIA/102212/2008, High-Performance Computing over the Large-Scale Internet.

REFERENCES

1. Trefethen LN, Bau D. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics: Philadelphia, PA, 1997.
2. Lenstra AK, Lenstra Jr. HW, Manasse MS, Pollard JM. The number field sieve. In *The Development of the Number Field Sieve*, vol. 1554, Lenstra AK, Lenstra Jr. HW (eds)., Lecture Notes in Mathematics. Springer-Verlag: Berlin, 1993; 11–42.
3. Montgomery PL. A block Lanczos algorithm for finding dependencies over GF(2). In *Advances in Cryptology—EUROCRYPT '95 (Saint-Malo, 1995)*, vol. 921, Guillou LC, Quisquater JJ (eds)., Lecture Notes in Computer Science. Springer-Verlag: Berlin, 1995; 106–120.
4. Wiedemann DH. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory* 1986; **32**:54–62.
5. Coppersmith D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation* 1994; **62**:333–350.
6. Cvetkovic DM, Doob M, Sachs H. *Spectra of Graphs* (3rd edn). Johann Ambrosius Barth Verlag: Heidelberg-Leipzig, 1995.
7. Aoki K, Shimoyama T, Ueda H. Experiments on the linear algebra step in the number field sieve. In *IWSEC*, vol. 4752, Miyaji A, Kikuchi H, Rannenberg K (eds)., Lecture Notes in Computer Science. Springer: Okinawa, Japan, 2007; 58–73.
8. Kleinjung T, Aoki K, Franke J, Lenstra AK, Thomé E, Bos JW, Gaudry P, Kruppa A, Montgomery PL, Osvik DA, Te Riele H, Timofeev A, Zimmermann P. Factorization of a 768-bit RSA modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10. Springer-Verlag: Berlin, Heidelberg, 2010; 333–350. Available from: http://portal.acm.org/citation.cfm?id=1881412.1881436 [Accessed on 20 January 2014].
9. Kleinjung T, Nussbaum L, Thomé E. Using a grid platform for solving large sparse linear systems over GF(2). *11th ACM/IEEE International Conference on Grid Computing (Grid 2010)*, Brussels Belgium, 2010; 161–168. Available from: http://hal.archives-ouvertes.fr/inria-00502899/PDF/grid.pdf [Accessed on 20 January 2014].
10. Courtois N, Klimov A, Patarin J, Shamir A. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'00. Springer-Verlag: Berlin, Heidelberg, 2000; 392–407. Available from: http://dl.acm.org/citation.cfm?id=1756169.1756206 [Accessed on 20 January 2014].
11. Yang BY, Chen OCH, Bernstein DJ, Chen JM. Fast software encryption, Biryukov A (ed.). Springer-Verlag: Berlin, Heidelberg, 2007; 290–308. Available from: http://dx.doi.org/10.1007/978-3-540-74619-5_19 [Accessed on 20 January 2014].
12. Cheng CM, Chou T, Niederhagen R, Yang BY. Solving quadratic equations with XL on parallel architectures. In *CHES*, vol. 7428, Prouff E, Schaumont P (eds)., Lecture Notes in Computer Science. Springer: Leuven, Belgium, 2012; 356–373. Available from: http://dx.doi.org/10.1007/978-3-642-33027-8_21 [Accessed on 20 January 2014].

13. Vuduc R, Demmel JW, Yelick KA, Kamil S, Nishtala R, Lee B. Performance optimizations and bounds for sparse matrix-vector multiply. *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Baltimore, MD, USA, 2002; 1–35. Available from: http://portal.acm.org/citation.cfm?id=762822 [Accessed on 20 January 2014].
14. Im EJ. Optimizing the performance of sparse matrix-vector multiplication. *Ph.D. Thesis*, EECS Department, University of California, Berkeley, 2000. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2000/5556.html [Accessed on 20 January 2014].
15. Buluç A, Fineman JT, Frigo M, Gilbert JR, Leiserson CE. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09. ACM: New York, NY, USA, 2009; 233–244. Available from: http://doi.acm.org/10.1145/1583991.1584053 [Accessed on 20 January 2014].
16. Pothen A. Graph partitioning algorithms with applications to scientific computing. *Technical Report*, Old Dominion University: Norfolk, VA, USA, 1997.
17. Im EJ, Yelick KA. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01. Springer-Verlag: London, UK, UK, 2001; 127–136. Available from: http://portal.acm.org/citation.cfm?id=645455.653756 [Accessed on 20 January 2014].
18. Fog A. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2012. Available from:http://www.agner.org/optimize/#manuals [Accessed on 20 January 2014].
19. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B & 2C: Instruction Set Reference. 325383-044*, 2013. Available from: http://download.intel.com/products/processor/manual/325383.pdf [Accessed on 20 January 2014].
20. Vuduc RW. Automatic performance tuning of sparse matrix kernels. *Ph.D. Thesis*, University of California, Berkeley, December 2003.
21. Im EJ, Yelick KA, Vuduc R. SPARSITY: framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications* February 2004; **18**(1):135–158.
22. Vuduc R, Demmel JW, Yelick KA. OSKI: a library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series. Institute of Physics Publishing: San Francisco, CA, USA, 2005; 521–530.
23. Boyer B, Dumas JG, Giorgi P. Exact sparse matrix-vector multiplication on GPU's and multicore architectures. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10. ACM: New York, NY, USA, 2010; 80–88. Available from: http://doi.acm.org/10.1145/1837210.1837224 [Accessed on 20 January 2014].
24. Stach P. Optimizations to NFS Linear Algebra. *CADO Workshop on Integer Factorization*, Villers-lès-Nancy, France, 2008. Available from: http://cado.gforge.inria.fr/workshop/abstracts.html [Accessed on 20 January 2014].
25. Williams R. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *SODA*, Bansal N, Pruhs K, Stein C (eds). SIAM: New Orleans, Louisiana, USA, 2007; 995–1001.
26. Arlazarov VL, Dinic EA, Kronrod MA, Faradzev IA. On economical construction of the transitive closure of an oriented graph. *Soviet Mathematics Doklady* 1970; **11**:1209–1210.
27. Bernstein DJ. Optimizing linear maps modulo 2. *Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers*, Berlin, Germany, 2009; 3–19.
28. Lupanov OB. On rectifier and contact-rectifier circuits. *Doklady Akademii Nauk SSSR* 1956; **111**:1171–1174.
29. Granlund T, Montgomery PL. Division by Invariant Integers using Multiplication. In *PLDI*, Sarkar V, Ryder BG, Soffa ML (eds). ACM: Orlando, Florida, USA, 1994; 61–72. Available from: http://doi.acm.org/10.1145/178243.178249 [Accessed on 20 January 2014].
30. Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications* August 2000; **14**:189–204. Available from: http://portal.acm.org/citation.cfm?id=1057150.1057156 [Accessed on 20 January 2014].