A code-based sparse binary matrix representation

Name1
Affiliation1
Email1

Name2
Affiliation2
Email2

Abstract

Sparse matrix-vector multiplication dominates the performance of many scientific and industrial problems. For example, iterative methods for solving linear systems rely on the performance of this critical operation. The particular case of binary matrices shows up in many important areas of computing, such as graph theory and cryptography. Unfortunately, irregular memory access patterns cause poor memory throughput, slowing down this operation.

We transform the matrix into a straight-line program that takes full advantage of the instruction cache. The regular loop-less pattern of the program minimizes 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. When compared to a Compressed Row Storage (CRS) implementation, we obtain a 20% performance improvement in a binary sparse matrix with 5426753 rows and weight 370909586.

Keywords `x86_64`, number field sieve, sparse matrix-vector multiplication

1. Introduction

Many problems in areas such as cryptography, graph theory and pattern recognition [1, 20, 27] can be modeled as linear algebra problems. Solving those problems often involves large numbers of sparse matrix-vector multiplications (SMVM), one of the most fundamental operations in numerical linear algebra. For instance, the Lanczos [18] and Conjugate Gradient [11] methods to solve linear systems rely heavily on the speed of this operation, which may account for as much as 90% of total runtime.

The particular case of binary sparse matrices is also widely used in practice, including for cryptanalysis [1, 8, 31]. Integer factorization algorithms, such as the quadratic sieve [23] and the number field sieve [19], 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 [21] and Block Wiedemann [6], which require $O(n)$ matrix-vector multiplications, are the most popular algorithms. In a recent factorization record [16, 17], the matrix to solve was of dimension $192,796,550 \times 192,795,550$, with $27,797,115,920$ nonzero entries, totalling about 105 GB in disk space. It was solved using the Block Wiedemann method, which took roughly 119 days — 85% of it spent on sparse matrix-vector multiplications.

The bottleneck in sparse matrix-vector multiplication is often poor memory access locality, causing suboptimal memory bandwidth [29]. Many different representations try to push memory accesses closer together, to improve locality [2, 5, 12, 25]. One common characteristic of most, if not all, is that they only improve *data* cache accesses — instruction cache remains largely unused.

However, in most modern CPUs, like Intel's Core 2 and i7 processor lines and 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. With the representation we use, 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. Our focus in this paper is on *serial*, or single-core, speed and not (yet) with multi-core processor idiosyncrasies.

The approach we took in our work was to write a compiler to convert the sparse matrix, which represents an $\mathbb{F}_2^n \to \mathbb{F}_2^n$ linear map, into `x86_64` machine code implementing that very map. This has a number of advantages:

Full L1 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 cache, despite their single time utilization.

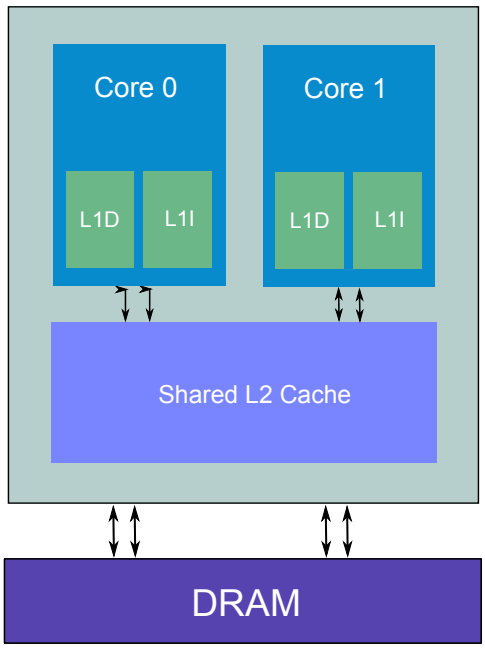

Figure 1. Core 2 memory layout.

Flexibility There is a large volume of published research on reordering, partitioning, and blocking techniques (e.g., [7, 13, 24]). Most of these improvements require changing the representation of the matrix, which also requires new code to use it. Using a linear map program allows us to freely integrate such techniques without having to change any extra matrix handling functionality. For example, a matrix in the Compressed Sparse Row (CSR) format [2], when converted to, say, Compressed Sparse Block (CSB) [5] format, will require new code to deal with the new representation. When one adds a new technique to our code representation, only the matrix compiler has to change, while everything else (i.e., the code that performs the matrix-vector multiplication) remains unchanged.

Implicit prefetching Modern processors have very elaborate predictive mechanisms to avoid pipeline stalls due to branching mispredictions, false dependencies, and so on [10]. 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.

We organize the rest of this paper as follows. Section 2 describes the architecture we targeted with our techniques. Section 3 describes our techniques to convert a CSR matrix into code. Finally, Section 4 describes and explains our experimental results, and Section 5 discusses the results and concludes with future directions.

2. Target architecture

In this paper, we target the `x86_64` architecture, as seen in the Intel Core 2 processor line. For this reason, we must start by overviewing the precise architecture we target. Nevertheless, we must emphasize that our idea is not restricted to this specific architecture. To create machine code for other microarchitectures available today, such as the Core i7 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 Core 2 pipeline can be divided in several main stages: instruction fetching, decoding, dispatching, execution and retirement.

The instruction fetching in the Core 2 has a total bandwidth of 16 bytes per clock, after which it passes through the predecoder, a mechanism that detects where instructions begin (due to the variable-sized instruction set). The combined throughput of the fetching and predecoding is 16 bytes or 6 instructions, whichever is the smallest [10]. The optimal instruction length is roughly 3 bytes for the best possible fetching throughput.

The instruction decoder in the Core 2 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 ran by the execution units in the next stages.

After decoding, instructions are renamed, if needed, and sent to a reservation station. Here, the Core 2 implements dynamic scheduling through Tomasulo's algorithm [26] with 5 execution units: 3 ALUs, 1 memory read unit, 1 ALU for address calculation, and 1 memory write unit. Thus, at any given time, it is possible to execute at most 6 μops per cycle, one less than the decoder is able to output.

2.2 Memory system

The Core 2, like most recent CPUs, spends most of its area on caches. It sports 2 32KB 8-way set associative L1 caches, exclusive to each core, and a larger (usually) 4MB 16-way set associative L2 cache, shared between two cores (cf. Figure 1). Each access to the L1 caches costs 3 cycles, while the L2 has a latency of 15 cycles. In each cycle, the L1 caches are able to pull 32 bytes from the L2 caches.

There are 3 hardware L1 prefetchers (per core) and 2 L2 prefetchers in the Core 2. The 3 L1 prefetchers are divided into 1 instruction prefetcher and 2 data prefetchers. These prefetchers detect access patterns, and preload the caches with data likely to be used.

3. Fast sparse matrix-vector multiplication

In our struggle to decrease the running time to solve sparse matrices created from integer factorization algorithms, we

```
void spmv(const u32 *colind, const u32 *rowidx,
     const u32 nrows, const u32 *in, u32 *out)
{
    u32 i,j;
    u32 *ptr = colind;
    for(i=0; i < nrows; ++i)
    {
        const u32 ncols = rowidx[i+1]-rowidx[i
            ];
        u32 s = 0;
        for(j=0; j < ncols; ++j)
            s ^= in[*colind++];
        out[i] = s;
    }
}
```

Figure 2. A typical CRS matrix-vector multiplication implementation.

found two main options: the Block Lanczos and the Block Wiedemann algorithms. The Lanczos approach has an heuristic cost of $\frac{n}{b-0.73}$ matrix-vector multiplications, b being the blocking factor, and Wiedemann has a running time of $3(\frac{n}{b} + \frac{n}{b})$. Lanczos, however, requires that input matrices be symmetric — this, in turn, forces the use of $\frac{n}{b-0.73}$ matrix-vector multiplications *and* $\frac{n}{b-0.73}$ transposed matrix-vector multiplications. The Wiedemann approach is also distributable to several different sites for a key step [17], which makes it more desirable for very large jobs.

3.1 Sparse matrix-vector multiplication

A common way to represent (binary) sparse matrices is Compressed Row Storage (CRS) [2]. In this representation, there are 3 arrays. One array contains the indexes 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. Figure 2 exhibits a simple implementation of sparse matrix-vector multiplication, with the binary matrix in CRS representation.

The major problem with CRS and sparse matrix-vector multiplication is memory bandwidth. Since accesses to the `in` vector (cf. Figure 2) are potentially highly irregular, memory latency is often the bottleneck [29]. Many techniques have been proposed over the years, most notably register blocking, cache blocking, software pipelining, software prefetching, TLB caching, among others [12, 14, 15, 22, 28, 30].

Consider the following 5×5 matrix

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

implementing the linear map

$$\mathbf{A} \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} = \begin{pmatrix} c+e \\ a+c+d \\ e \\ a \\ b+c+d \end{pmatrix} \tag{2}$$

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. [3]). As such, we do the optimization and compilation of the matrix ourselves. This section describes the various approaches and improvements possible on the target (`x86_64`) architecture; knowledge of this instruction set is assumed.

3.2 Base approach

Our initial approach converts this mapping to `x86_64` code in the most straightforward way possible: simply convert the arithmetic of a normal CRS matrix-vector multiplication (cf. Figure 2) to an actual straight-line function, callable from most native compiled languages such as C or FORTRAN. We use 32-bit registers whenever possible, to avoid the `REX.W` prefix necessary for 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.

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 3 shows the `x86_64` assembly code[1] obtained by our initial approach.

3.3 Reducing code size

Once we have the initial compiler, the first 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 bytes (e.g., `MOV EAX, [RSI+32]`), the instruction requires 3 bytes. Finally, for general 32-bit displacements (e.g., `MOV EAX, [RSI+12345678]`), the instruction requires 6 bytes. In Figure 3, for instance, all but one access are 3 bytes, since $\mathbf{A}$ is small enough that the vector fits 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

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

```
; void spvm(u32 *out, u32 *in)
; RSI contains source, RSI 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
retn                ; C3
```

Figure 3. Straight-line program implementing the linear mapping represented by matrix $\mathbf{A}$.

```
; 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 4. Straight-line program for $\mathbf{A}$ using basic register blocking ($B = 4$).

are at least 3 nonzero entries in a 256-byte interval. Since we are changing RSI, we must reset to its original value in the beginning of each row: we do this using the PUSH and POP instructions, each costing 1 byte 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 get $3n + 3\omega + \frac{6}{64}\omega$ bytes.

3.4 Register blocking

Register blocking is a technique already known from sparse linear algebra optimization [12], 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., [12]), code register blocking does not add fill overhead due to storing extra 0s to make the blocks dense.

We have implemented 2 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 4 registers used for the vector addresses, stack, and a temporary variable. Figure 4 shows the assembly code for matrix $\mathbf{A}$ with register blocking ($B = 4$).

To store row results, we still prefer to use the STOSD instruction. Since 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 4). 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 3.2) uses $2n + 6\omega$ bytes, and the $B = 13$ variant uses $\frac{2\cdot4+3\cdot9}{13}n + \frac{4\cdot6+9\cdot7}{13}\omega$ bytes. In the best case, i.e., when every row has the same entries, we get $4n + 3n + 2\omega$ and $4n + 3n + \frac{2\cdot4+3\cdot9}{13}\omega$ bytes, for $B = 4$ and $B = 13$ respectively.

3.5 Sorted register blocking

From the example in Figure 4, it is easy to see that the basic register blocking method of Section 3.4 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 blindingly converting the matrix operations as they appear on its CRS representation, one can instead sort the order of operations by memory accesses. Since 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

```
;
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 5. Straight-line program for $\mathbf{A}$ 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 6. Straight-line program implementing the linear mapping represented by matrix $\mathbf{A}$, with non-temporal stores.

speed advantages: memory accesses become optimal (up to block size), and there is an increased likelihood of existing a neighborhood of nonzero entries less than 256 bytes apart. Figure 5 shows the assembly listing for the sorted register blocking output of matrix $\mathbf{A}$.

3.6 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.

To avoid cache pollution, we employ the SSE2 instruction `MOVNTI`. This instruction is one of several *non-temporal* store operations provided by the SSE2 instruction set. Non-temporal instructions give a hint to the processor that the stored data is 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 3.2, this tweak slightly increases the code size to at most $4n + \frac{6}{64}n + 6\omega$ bytes. Figure 6 displays the code for $\mathbf{A}$ when using `MOVNTI`.

3.7 Extended register blocking

In Section 3.4, 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 blocks. Figure 7 illustrates the workings of extended register blocking.

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 3.2 would put the best case scenario of the code size at $n + 18\omega$, and the worst case at $n + 21\omega$. Thus, we do not use this method for large matrices. In comparison, using the stack memory to store temporary elements would cost $n + 12\omega$.

3.8 Memory prefetching

When one converts a sparse matrix into code, it becomes mostly unnecessary to try and prefetch the matrix itself — 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, since 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, `PREFETCH` for AMD) are not short, however: each prefetch instruction costs about 7 bytes, so it is used sparsely.

3.9 Summary

We have studied several possible combinations of `x86_64` operations previously in this section. Table 1 summarizes their relative advantages and disadvantages, in terms of code size. To recap, n is the number of rows in the matrix, and ω is its weight. All sizes after the initial case from Section 3.2 are relative to each technique implemented individually on top of this base case, not cumulatively.

```
; Suppose we want to access register number 33
    of the register block
pextrd eax, xmm8, 1 ; 33 div 4, 33 mod 4
xor eax, [esi + 40]
pinsrd xmm8, eax, 1 ; Insert back into register
     block
```

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}

Figure 7. How to use the XMM register set as an array of addressable 32-bit integers.

Method(s)	Best case	Worst case
CSR	$4n + 4\omega$	$4n + 4\omega$
Section 3.2	$n + 6\omega$	$n + 6\omega$
Section 3.3	$3n + 3.09\omega$	$5n + 6\omega$
Section 3.4 $B = 4$	$4n + 2n + 2\omega$	$2n + 6\omega$
Section 3.4 $B = 13$	$4n + 2.69n + 2\omega$	$2.69n + 6.69\omega$
Section 3.5$B = 4$	$4n + 2n + 2\omega$	$2n + 6\omega$
Section 3.5$B = 13$	$4n + 2.69n + 2\omega$	$2.69n + 6.69\omega$
Section 3.6	$4.09n + 6\omega$	$4.09n + 6\omega$
Section 3.7	$n + 18\omega$	$n + 21\omega$

Table 1. Code sizes of the various approaches described in Section 3.

4. Results and discussion

To benchmark our code, we used two different matrices obtained from number field sieve factorization jobs. The first one, small.crs, resulting of a 265-bit Number Field Sieve (NFS) factorization, is $150,615 \times 150,802$ in size and $14,599,768$ in nonzero entries. The second matrix, large.crs, results from a 512-bit NFS factorization as well, and is $5,426,753 \times 5,426,928$ with $370,909,586$ nonzero entries. Due to size concerns, we did not test the extended register blocking of Section 3.7, as we did not have enough memory. The test machine was an Intel Core 2 Duo E8400, using DDR2 RAM running at 800 MHz (CL5).

We test our CRS implementation in $\mathbb{F}_2$ against 5 combinations of the techniques of Section 3:

Method 1 The base method of Section 3.2;

Method 2 Method 1, plus RSI movement;

Method 3 Method 2, plus $B = 4$ sorted register blocking;

Method 4 Method 2, plus $B = 12$ sorted register blocking;

Method 5 Method 4, plus non-temporal stores.

Our first tests concern our key performance figures: time (measured in clock cycles) and code size. Code size is measured in bytes per nonzero matrix entry. Table 2 lists those performance figures when measured for the small.crs matrix, roughly 45MB in size when stored in CRS format. In this matrix, our code is always faster than the CRS implementation, including when using the worst of our approaches, i.e., the base case of Section 3.2.

In the large.crs matrix, the situation is slightly different, as shown in Table 3. This 1.5GB matrix puts far more pressure into every part of the memory subsystem, and our more naïve implementations fall behind 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 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 5) — in average, the register blocked code (with RSI updates) generates 2 and 3 byte instructions, the optimal size on the Core 2 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 20% less cycles than CRS, despite being $1.52\times$ 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 PAPI library [4]. Figure 8 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 nearly 0 instruction cache misses. This is natural, since even an optimized CRS implementation will generally have a small enough loop to fit within Core 2's loopback buffer [10]. Our implementation has a relatively small number of instruction cache misses, which is to be expected due to the large code size. Its other (L1d and L2) cache misses, however, are both lower than CRS; the total sum of misses of our method saves over 150 million cache misses overall, which results in its superior performance.

5. Conclusion

In this paper, we have studied the implications and consequences of fully converting a binary sparse matrix to code. Our method takes full advantage of the L1 instruction cache, and sorted register blocking allied with non-temporal hints and careful size-coding allowed us to produce straight-line

Method	CRS	Method 1	Method 2	Method 3	Method 4	Method 5
Clock cycles	60,430,464	51,368,463	51,153,390	46,285,497	45,591,255	44,175,024
Bytes/nonzero	4.0264	6.01392	5.71479	5.30643	5.61708	5.66502

Table 2. Speeds (in clock cycles) and bytes per nonzero entry for the `small.crs` matrix.

Method	CRS	Method 1	Method 2	Method 3	Method 4	Method 5
Clock cycles	17,841,138,813	18,960,412,932	18,898,653,474	16,471,466,667	14,371,353,612	14,318,379,171
Bytes/nonzero	4.0245	6.0146	5.88905	5.63507	6.09096	6.13851

Table 3. Speeds (in clock cycles) and bytes per nonzero entry for the `large.crs` matrix.

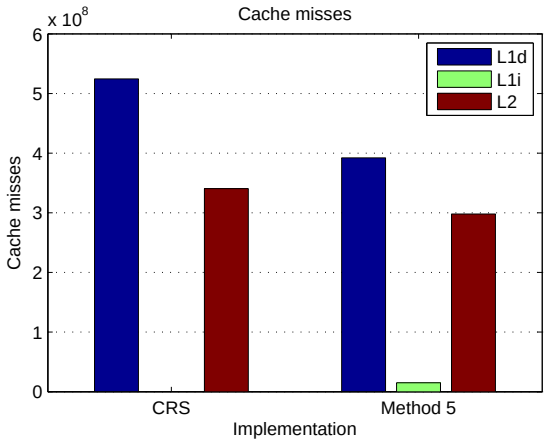

Figure 8. Comparison of cache misses between the CRS implementation and Method 5.

programs representing the same linear map in relatively small space, and up to 20% superior performance.

While our focus was on the arithmetic of $\mathbb{F}_2$ matrices for Block Wiedemann in integer factorization, our ideas and methods have broader applications. Many matrices resulting from adjacency graphs and web 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 `MOVAPS` or `MOVAPD`. Most size considerations should remain similar after this conversion.

Some questions have been still left unanswered. We have yet only studied the performance of our code on the single-core single-threaded scenario; since today's machines have several cores, this is a very interesting (and active!) research direction.

References

[1] K. Aoki, T. Shimoyama, and H. Ueda. Experiments on the Linear Algebra Step in the Number Field Sieve. In A. Miyaji, H. Kikuchi, and K. Rannenberg, editors, *IWSEC*, volume 4752 of *Lecture Notes in Computer Science*, pages 58–73. Springer, 2007.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] B. Boyer, J.-G. Dumas, and P. Giorgi. 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, pages 80–88, New York, NY, USA, 2010. ACM.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14:189–204, August 2000.

[5] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. 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, pages 233–244, New York, NY, USA, 2009. ACM.

[6] D. Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Mathematics of Computation*, 62:333–350, 1994.

[7] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[8] F. Didier. Using wiedemann's algorithm to compute the immunity against algebraic and fast algebraic attacks. In R. Barua and T. Lange, editors, *INDOCRYPT*, volume 4329 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2006.

[9] U. Drepper. What Every Programmer Should Know About Memory. `http://people.redhat.com/drepper/cpumemory.pdf`, 2007.

[10] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `http://www.agner.org/optimize/#manuals`, January 2011.

[11] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of*

the National Bureau of Standards, 49(6):409–436, December 1952.

[12] E.-J. Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, EECS Department, University of California, Berkeley, Jun 2000.

[13] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Science*, volume 2073 of *LNCS*, pages 127–136, San Francisco, CA, May 2001. Springer.

[14] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[15] A. Jain. pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures. Master's thesis, UC Berkeley, July 2008.

[16] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 333–350, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] T. Kleinjung, L. Nussbaum, and E. Thomé. Using a grid platform for solving large sparse linear systems over GF(2). In *11th ACM/IEEE International Conference on Grid Computing (Grid 2010)*, Brussels Belgium, 10 2010.

[18] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[19] A. K. Lenstra, J. Hendrik W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. In A. K. Lenstra and J. Hendrik W. Lenstra, editors, *The development of the number field sieve*, volume 1554 of *Lecture Notes in Mathematics*, pages 11–42, Berlin, 1993. Springer-Verlag.

[20] K. Levchenko, J. Ma, Z. Xiao, and Y. Zhang. Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space. Technical Report CS2006-0866, UCSD, September 2006.

[21] P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In L. C. Guillou and J.-J. Quisquater, editors, *Advances in cryptology—EUROCRYPT '95 (Saint-Malo, 1995)*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120, Berlin, 1995. Springer-Verlag.

[22] R. Nishtala, R. Vuduc, J. Demmel, and K. Yelick. When Cache Blocking Sparse Matrix Vector Multiply Works and Why. *Applicable Algebra in Engineering, Communication and Computing*, March 2007.

[23] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Advances in cryptology: EUROCRYPT '84*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182, Berlin, 1985. Springer-Verlag.

[24] A. Pothen. Graph Partitioning Algorithms with Applications to Scientific Computing. Technical report, Norfolk, VA, USA, 1997.

[25] Y. Saad. *SPARSKIT: A basic tool kit for sparse matrix computations*. University of Minnesota Department of Computer Science and Engineering, 200 Union Street S.E., Minneapolis, MN 55455 USA (612) 624 7804, June 1994.

[26] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[27] M. v. Uitert, W. Meuleman, and L. Wessels. Biclustering sparse binary genomic data. *Journal of Computational Biology*, 15(10):1329–1345, 2008.

[28] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[29] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proc. ACM/IEEE Conf. Supercomputing (SC)*, Baltimore, MD, USA, November 2002.

[30] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

[31] B.-Y. Yang, O. Chen, D. Bernstein, and J.-M. Chen. Analysis of QUAD. In A. Biryukov, editor, *Fast Software Encryption*, volume 4593 of *Lecture Notes in Computer Science*, pages 290–308. Springer Berlin / Heidelberg, 2007.