Generating Optimized Sparse Matrix Vector Product over Finite Fields

Pascal Giorgi[1] and Bastien Vialla[1]

LIRMM, CNRS, Université Montpellier 2 ,
`pascal.giorgi@lirmm.fr, bastien.vialla@lirmm.fr`

Abstract. Sparse Matrix Vector multiplication (SpMV) is one of the most important operation for exact sparse linear algebra. A lot of research has been done by the numerical community to provide efficient sparse matrix formats. However, when computing over finite fields, one need to deal with multi-precision values and more complex operations. In order to provide highly efficient SpMV kernel over finite field, we propose a code generation tool that uses heuristics to automatically choose the underlying matrix representation and the corresponding arithmetic.

Keywords: sparse linear algebra, finite fields, SpMV

1 Introduction

Modern sparse linear algebra is fundamentally relying on iterative approaches such as Wiedemann or Lanczos. The main idea is to replace the direct manipulation of a sparse matrix with its Krylov subspace. In such approach, the cost is therefore dominated by the computation of the Krylov subspace, which is done by successive matrix-vector products. Let $A \in \mathcal{F}^{n\times n}$ be a sparse matrix with $O(n \log^{o(1)} n)$ non zero entries where $\mathcal{F}$ is a finite field. The matrix-vector product $y = Ax$ where $y, x \in \mathcal{F}^n$ costs $O(n \log^{o(1)} n)$ operations in $\mathcal{F}$. We call this operation SpMV in the rest of this paper. SpMV is a particular operation in the linear algebra framework, since it requires as much memory accesses as arithmetic operations. Basically, one entry in the matrix contributes to the SpMV computation only once. Therefore, on modern processor where memory hierarchy has a larger impact then arithmetical operations, data access is a major challenge to reach good performances. This challenge has been widely studied by the numerical community, and led to many different matrix storage for floating point numbers.

Over finite fields the situation is slightly different. Basically, the underlying arithmetic is more complex. Indeed, modern processor does not provide efficient support for modular operations. Furthermore, finite fields can be large and then requiring multiple precision arithmetic. Our main concern is weather the numerical formats are still satisfying when computing over finite fields and which arithmetic strategy is the most suited to the particularity of SpMV. This question has

[1] This work has been supported by the Agence Nationale pour la Recherche under Grants ANR-11-BS02-013 HPAC, ANR-12-BS02-001 CATREL.

been already addressed in many papers, as in [3, 6], but no general optimization approach has been designed. In this paper, we propose a general framework which incorporates most of the optimization techniques for SpMV over finite field. We provide a software tool, available at `www.lirmm.fr/~vialla/spmv.html`, that emphasizes our approach for prime fields at any precision.

2 General optimization approach

It is well know that SpMV performance is limited by memory accesses. Indeed, the irregularity access of the $x[i]$'s during SpMV does not allow the processor to prefetch data to the cache memory in advance. In order to minimize cache misses, one need to minimize the memory footprint of the matrix while preserving a cache aware structure. To further speed-up SpMV over finite field, one need to minimize the number of clock cycle per arithmetic operation. This could be done by minimizing the modular reductions or taking care of particular entries in the matrix, i.e. ones and minus ones. One can evaluate *a priori* the impact of these optimizations by using the roofline model of [9].

Preprocessing the matrix in advance is a key tool to detect the most suited optimization. This can be done at runtime, as in the OSKI library [8]. Our proposed approach is to do this at compile time through two steps: 1) preprocessing the matrix to provide an optimization profile.; 2) generating an optimized SpMV with this profile.

3 Optimized SpMV generator

The workflow of our generator is given in Figure 1. It receives a matrix A and a prime number p such that the SpMV with A is performed over $\mathbb{F}_p$. Depending on the prime p and some characteristics of A, such as the number of ± 1 or the dimension, the generator choose the best suited matrix format and an arithmetic strategy. Then, it generates an optimization profile that can be used to compile a SpMV implementation for A over $\mathbb{F}_p$.

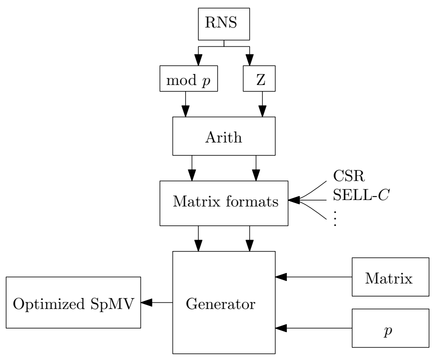

Fig. 1. Generator workflow

3.1 Matrix formats

A lot of matrix format have been proposed by the numerical community for SpMV: e.g. CSR, COO, BlockCSR[8], Compress Sparse Block [4]. In general, matrices over finite fields do not have any structural properties than can be used to improve performances. Therefore, we choose to focus on the CSR and

SELL-C format [7], and some adaptation avoiding the storage of ±1. However, our approach is generic and more format can be added if necessary.

The CSR format compress data according to the row indices. It needs three arrays: `val[]` for the matrix entries; `idx[]` for the column indices; `ptr[]` for the number of non-zero entries per row. The SELL-C format is a variant of CSR which is designed to incorporate SIMD operations. It sorts the rows according to their sparsity and split the matrix by chunk of size C. Each chunk is padded with zeros such that each row in a same chunck has exactly the same sparsity. The parameter C is a chosen to be a multiple of the SIMD unit's width. In order to minimize the memory footprint, our generator adapt the data types of every arrays, e.g. 4 bytes for `idx[]` when column dimension is $< 65\,536$. It also choose a data type related to $||A||_\infty$ rather than p for `val[]`.

3.2 Delayed modular reduction

As demonstrated in [5], performing modular reductions only when necessary leads to better performances. Hence, the computation is relaxed over the integer and needs that no overflow occurs.

Depending on the finite field, our generator will compute *a priori* the maximum value of k such that $k(p-1)||A||_\infty$ does not overflow. Note that knowing k at compile time will allow the compiler to perform loop unrolling. As delayed modular reduction is fundamentally tied with the underlying data type, our approach is to use the best suited one to reduce the number of modular reductions. Nevertheless, some compromises must be done between the cost of the standard operations $(+, \times)$ *vs* the number of reductions.

3.3 Hybrid

In most applications over finite fields, many matrix entries are ±1. Thus, one can avoid superfluous multiplication within SpMV and further reduce the memory footprint of the matrix. This approach have been developed in [3] using a splitting of the matrix A in 3 matrices: A_1 storing only 1's , A_{-1} storing -1's and finally, A_λ store the rest of the entries. SpMV is then computed independently for each matrices and the results are sum up, i.e. $y = Ax = A_1x + A_{-1}x + A_\lambda x$.

The drawback of this method is to amplify the number of cache misses arising during the reading of the vector x. Indeed, most of the matrices have a spatial locality in their row entries which is useful to avoid cache misses.

Our proposed hybrid approach is to keep this spatial locality such that SpMV still can be performed row by row. CSR format is well designed for this approach. Indeed, for each row we can store in `idx[]`, the column indices of the entries different ±1, then the indices of ones and the minus ones. We can do exactly the same for `ptr[]`, and `val[]` only stores the entries different from ±1. We call this hybrid format CSRHyb. Note this approach, cannot be directly applied to SELL-C since the zero padding may introduce too much memory overhead.

To circumvent this, one must store entries different from ±1 in SELL-C format and the ±1 in CSR format, but these two formats must be interleaved by row.

4 Benchmarks

Our benchmarks have been done on matrices arising in mathematical applications. The Table 1 gives the characteristic of such matrices (available at `http://hpac.imag.fr`). In this table, nnz is the number of non zero entries, nnz_{row} is the average number of non zero entries in a row, and k_{max} is the maximum number of non zero entries in a row.

Name	Dimensions	nnz	nnz$_{row}$	k_{max}	±1	$\neq\pm1$	Problems
cis.mk8-8.b5	564 480 × 376 320	3.3M	6	6	3.3M	0	(A)
GL7d17	1 548 650 × 955 128	25M	16	69	25M	382K	(B)
GL7d19	1 911 130 × 1 955 309	37M	19	121	36M	491K	(B)
GL7d22	349 443 × 822 922	8.2M	23	403	7M	307K	(B)
M06-D9	1 274 688 × 1 395 840	9.2M	7	10	9.2M	0	(E)
rel9	5 921 785 × 274 669	23M	3	4	23M	19K	(C)
relat9	9 746 231 × 549 336	38M	3	4	23M	29K	(C)
wheel_ 601	902 103 × 723 605	2.1M	2	602	38M	29K	(F)
ffs619	653 365 × 653 365	65M	100	413	60M	5M	(D)
ffs809	3 602 667 × 3 602 667	360M	100	452	335M	25M	(D)

(A) Simplicial complexes from homology; (B) Differentials of the Voronoï complex of the perfect forms (C) Relations;(D) Function field sieve ; (E) Homology of the moduli space of smooth algebraic curves $M_{g,n}$; (F) Combinatorial optimization problems.

Table 1. List of matrices arising in mathematical applications

We used g++ 4.8.2 and an Intel bi-Xeon E5-2620, 16GB of RAM for our benchmarks. We performed a comparison with the best SpMV available at this time in the LinBox library[1] (rev 4901) based on the work of [3].

4.1 Prime field $\mathbb{F}_p$with small p

In this section we consider the case where $(p-1)^2$ fits the mantissa of a double floating point number, e.g. 53 bits. In this case, the modular reduction is costly compare to the standard operations. However, it does not worth it to extend the precision beyond 53 bits to avoid most of the reductions. Our strategy is then to use `double` and to find the largest k such that $k||A||_\infty(p-1) < 2^{53}$ and perform reduction at least every k entries in a row. If the matrix does not have too many ±1, the SELL-C format will be chosen to better exploit the SIMD vectorization, otherwise the CSRHyb format will be preferred.

The Figure 2 gives the relative performances of our optimized SpMV against the one of LinBox for the prime field $\mathbb{F}_{1048583}$. One can see that our code is always faster than the CSR implementation of LinBox, up to a speed-up of 2.2.

[1] www.linalg.org

Indeed, this can be explain by the fact that most matrices have a many ± 1 entries and that the CSR of LinBox is not handling such particularity. The implementation of the hybrid format from [3] is not yet fully operational in LinBox and we did not get the chance yet to compare to it. However, following the speed-up of the hybrid format *vs* the CRS one given in [3, Figure 3], which is less than 1.5, we are confident in the performance of our optimized SpMV.

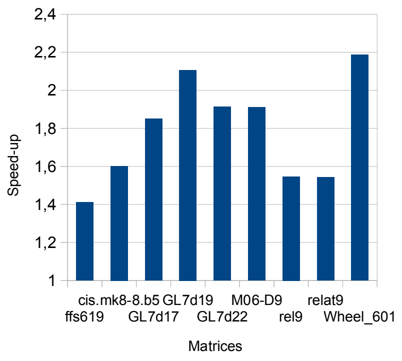

Fig. 2. Speed-up of our generated SpMV against LinBox over $\mathbb{F}_{1048583}$.

4.2 Prime field $\mathbb{F}_p$with p multiple precision

Our motivations come primarily from the computation of discrete logarithms over finite fields [1]. We focus only on matrices which have small entries compare to the prime p, e.g. less than a machine word, since it is mostly the case in mathematical applications.

In order to compute with multiple precision integers, one can use the well known GMP library [2] which is the fastest one for each single arithmetic operations. However, when dealing with vectors of small integers, e.g. $\approx$ 1024 bits, the GMP representation through pointer makes it difficult to exploit cache locality. In such a case, one should prefer to use a fixed precision representation through a residue number system [2], called RNS for short. Such approach provides intrinsic data locality and parallelism which are good for SpMV and its SIMD vectorization. The difficulty is then transferred to the reduction modulo p that cannot be done in the RNS basis. However, one can use the explicit chinese remainder theorem [2, 6] to provide a reduction that can use SIMD instructions. Furthermore, one can use matrix multiplication to perform modular reduction of a vector of RNS values and then better exploit data locality and SIMD.

In order to minimize the memory footprint of the matrix, we propose to store the matrix entries in `double` and convert them to RNS on the fly. The m_i's are chosen so that $(m_i - 1)^2$ fits in 53-bits to allow floating point SIMD in the RNS arithmetic. Larger m_i's could be chosen to reduce the RNS basis but this would induce a more complex SIMD vectorization which makes it harder to reach sustainable performances. In our multiple precision SpMV, the m_i's are chosen so that $k_{max}(m_i - 1)^2 < 2^{53}$ and $M = \prod_i m_i > (p-1) \cdot k_{max} \cdot ||A||_\infty$. This both ensures that the modular reduction mod m_i and mod p can be done only once per row. The matrix format is CSRHyb since the SIMD vectorization is done over the arithmetic rather then being on SpMV operations.

[2] https://gmplib.org/

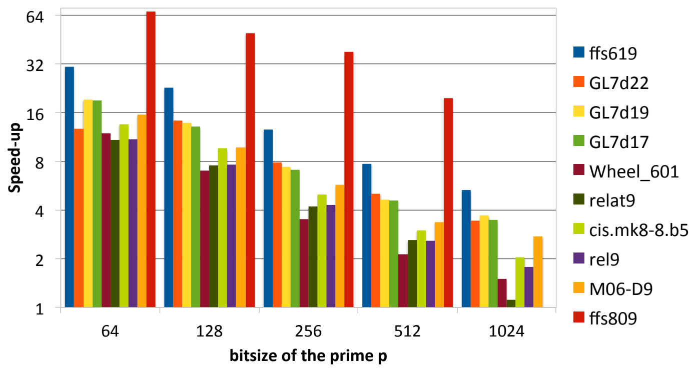

Fig. 3. Speed-up of our generated multiple precision SpMV over LinBox.

The Figure 3 gives the relative performances of our optimized SpMV against the one of LinBox for prime fields of bitsize, 64, 128, 256, 512, and 1024. One can see that our code is always faster than LinBox one, up to a speed-up of 67. Indeed, this can be explain by the fact that most matrices have many ± 1 entries. But, mainly because LinBox stores matrix entries as GMP integers while our SpMV stores them as `double`. This has two consequences on LinBox SpMV performances. First, matrix entries are not store contiguously and then many cache misses are done. Secondly, LinBox use the arithmetic of GMP which is not using any SIMD vectorizations for mixed precision arithmetic.

References

1. R. Barbulescu, C. Bouvier, J. Detrey, P. Gaudry, H. Jeljeli, E. Thomé, M. Videau, and P. Zimmermann. Discrete logarithm in GF(2^{809}) with FFS. In *Proc. of 17th International Conference on Practice and Theory in Public-Key Cryptography*, 2014.
2. D. J. Bernstein. Multidigit modular multiplication with the explicit chinese remainder theorem. http://cr.yp.to/papers/mmecrt.pdf, 1995.
3. B. Boyer, J.-G. Dumas, and P. Giorgi. Exact sparse matrix-vector multiplication on gpu's and multicore architectures. In *Proc. of the 4th International Workshop on Parallel and Symbolic Computation*, pages 80–88, July 2010.
4. A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 721–733, 2011.
5. J.-G. Dumas, P. Giorgi, and C. Pernet. Dense Linear Algebra over Finite Fields: the FFLAS and FFPACK package. *ACM Trans. Math. Soft.*, 35:19:1–19:42, 2008.
6. H. Jeljeli. Accelerating iterative spmv for discrete logarithm problem using gpus. hal-00734975, 2013. http://hal.inria.fr/hal-00734975/en/.
7. M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A.-R. Bishop. A unified sparse matrix data format for modern processors with wide simd units. arXiv:1307.6209, 2013. http://arxiv.org/abs/1307.6209.
8. R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. of SciDAC 2005*, Journal of Physics: Conference Series, pages 521–530, June 2005.
9. S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.