HAL
open science

Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures

Brice Boyer, Jean-Guillaume Dumas, Pascal Giorgi

To cite this version:

Brice Boyer, Jean-Guillaume Dumas, Pascal Giorgi. Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures. PASCO'10: 4th International Symposium on Parallel Symbolic Computation, Jul 2010, Grenoble, France. pp.80-88, 10.1145/1837210.1837224 . hal-00475185

HAL Id: hal-00475185

https://hal.science/hal-00475185

Submitted on 21 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures

Brice Boyer* Jean-Guillaume Dumas*† Pascal Giorgi‡

April 21, 2010

Abstract

We propose different implementations of the sparse matrix–dense vector multiplication (SpMV) for finite fields and rings $\mathbf{Z}/m\mathbf{Z}$. We take advantage of graphic card processors (GPU) and multi-core architectures. Our aim is to improve the speed of SpMV in the LinBox library, and henceforth the speed of its black box algorithms. Besides, we use this and a new parallelization of the sigma-basis algorithm in a parallel block Wiedemann rank implementation over finite fields.

1 Introduction

Nowadays, personal computers and laptops are often equipped with multicore architectures, as well as with more and more powerful graphic cards. The latter ones can be easily programmable for a general purpose computing usage (Nvidia Cuda, Ati Stream, OpenCL). Graphic processors offer nowadays quite frequently superior performance on a same budget as their CPU counterparts. However, programmers can also efficiently use many-core CPUs for parallelization e.g. with the OpenMP standard.

On the numerical side, several libraries automatically tune the sparse matrix kernels [19, 20, 16] and recently some kernels have been proposed e.g. for GPU's [17, 2, 1]. In this paper we want to adapt those techniques for exact computations and we first mostly focused on $\mathbf{Z}/m\mathbf{Z}$ rings, with m smaller that a machine word.

The first idea is to use the numerical methods in an exact way as has been done for dense matrix operations [7]. For sparse matrices, however, the extraction of sparse matrices is slightly different. Also, over small fields some more

*\{Brice.Boyer,Jean-Guillaume.Dumas\}@imag.fr, Université de Grenoble, Laboratoire Jean Kuntzmann, UMR CNRS 5224. 51, rue des Mathématiques, BP 53X, 38041 Grenoble, France.

†Part of this work was done while the second author was visiting the Claude Shannon Institute and the University College Dublin, Ireland, under a CNRS grant.

‡Pascal.Giorgi@lirmm.fr, Université Montpellier 2, Laboratoire LIRMM, UMR CNRS 5506. F34095 Montpellier cedex 5, France.

dedicated optimizations (such as a separate format for ones and minus ones) can be useful. Finally, we want to be able to use both multi-cores and GPU's at the same time and the best format for a given matrix depends on the underlying architecture.

Therefore, we propose an architecture with hybrid data formats, user-specified or heuristically discovered dynamically. The idea is that a given matrix will have different parts in different formats adapted to its data or the resources. Also we present a "just-in-time" technique that allows to compile on the fly some parts of the matrix vector product directly with the values of the matrix.

We have efficiently implemented[1] "`Sparse Matrix-Vector` multiplication" (SpMV) on finite rings, together with the transpose product and iterative process to compute the power of a matrix times a vector, or a sequence of matrix products.

We also make use of this library to improve the efficiency of the block Wiedemann algorithm's of the LinBox[2] library. Indeed, this kind of algorithm uses block "black box" [13] techniques: the core operation is a matrix-vector multiplication and the matrix is never modified. We use the new matrix-vector multiplication library, together with a new parallel version of the sigma-basis algorithm, used to compute minimal polynomials [11, 8].

In section 2 we present different approaches to the parallelization of the SpMV operation, with the adaptation of numerical libraries (section 2.3), new formats adapted to small finite rings (section 2.4) together with our new hybrid strategy and their iterative versions (section 2.5). Then in section 3 we propose a new parallelization of the block Wiedemann rank algorithm in LinBox, via the parallelization of the matrix-sequence generation (section 3.1) and the parallelization of the matrix minimal polynomial computation (section 3.2).

2 Sparse-Vector Matrix multiplication

We begin with introducing some notations. In this section, we will consider a matrix A; the element at row i, column j is $A[i, j]$. The number `nbnz` is the number of non zeros elements in matrix A, it has `row` lines and `col` columns. If $\mathbf{x}$ and $\mathbf{y}$ are vectors, then we perform here the operation $\mathbf{y} \leftarrow A\,\mathbf{x} + \mathbf{y}$. The general operation $\mathbf{y} \leftarrow \alpha A\,\mathbf{x} + \beta\,\mathbf{y}$ can then be done by pre-multiplying $\mathbf{x}$ and $\mathbf{y}$ by α and β respectively. The `apply` operation in black box algorithms, or $\mathbf{y} \leftarrow A\,\mathbf{x}$, is performed by first setting $\mathbf{y}$ elements to zero. For further use in block methods, we also provide the operation $\mathbf{Y} \leftarrow \alpha A\,\mathbf{X} + \beta\,\mathbf{Y}$ where $\mathbf{X}$ and $\mathbf{Y}$ are sets of vectors.

2.1 Sparse Matrix Formats and Multiplication

Sparse matrices arise form various domains and their shapes can be very specific. Taking into consideration the structure of a sparse matrix can dramatically

[1] `https://ljkforge.imag.fr/projects/ffspmvgpu/`
[2] `http://linalg.org`

improve the performance of SpMV. However, there is no general storage format that is efficient for all kind of sparse matrices.

Among the most important storage formats is the `COO` (coordinate) format which stores triples. It consists of three vectors of size `nbnz`, named `data`, `colid` and `rowid`, such that `data[k]=` A `[rowid[k],colid[k]]`.

The `CSR` (compressed storage row) stores more efficiently the previous representation: the `rowid` field is replaced by a (`row`$+1$) long `start` vector such that if `start[`i`]` $\leqslant k <$ `start[`$i+1$`]`, then `data[`k`]`$= A$`[`i`,colid[`k`]]`. In other words, `start` indicates where a line starts and ends in the other two ordered fields.

The `ELL` (ELLpack) format stores data in a denser way: it has `data` and `colid` fields such that `data[`i, j_0`]`$= A$`[`i`,colid[`i, j_0`]]`, where j_0 varies between 0 and the maximum number of non zero elements on a line of A. One notices that these fields can be stored in row-major or column-major order. A variant is the `ELL_R` format that adds a `row` long `rownb` vector that indicates how many non zero entries there are per line.

The `DIA` (DIAgonal) is used to store matrices with non zero elements grouped along diagonals. It stores these diagonals in an array along with offsets where they start. We refer to [1],[17] for more details on these formats.

This very schematic description of a few well-known formats shows that each of them has pros and cons. Our aim is to produce a more efficient implementation of the SpMV operation on finite fields than the one present in LinBox, taking first advantage of this variety of formats.

2.2 Finite field representation

We present now how the data is stored. We use data types such as `float`, `int`. Firstly, when doing modular linear algebra, we try to minimize the number of costly $fmod$ (reduction) operation calls. For instance, we prefer if possible the left loop to the right one in the next figure:

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
}
y = fmod(y,m);
```

```
for (i=0 ; i<n ; ++i){
    y += a[i] * b[i] ;
    y = fmod(y,m);
}
```

In this case, suppose $y = 0$ and $a[i]$, $b[i]$ are reduced modulo m at first, and M is the largest representable integer. Say that on $\mathbf{Z}/m\,\mathbf{Z}$, we represent the ring on $[\![0, m-1]\!]$. Then we can do at most M/m^2 such accumulations before reducing. We can also represent the ring on $[\![-\lfloor\frac{m-1}{2}\rfloor, \lceil\frac{m-1}{2}\rceil]\!]$. The latter representation enables us to perform twice more operations before a reduction, but this reduction is slightly more expensive. Another trade-off consists in choosing a `float` representation instead of `double` (on the architectures that support `double`). Indeed, operations can be much faster on `float` that on `double` but the `double` representation lets us do more operations before reduction. This is particularly true on some GPU's.

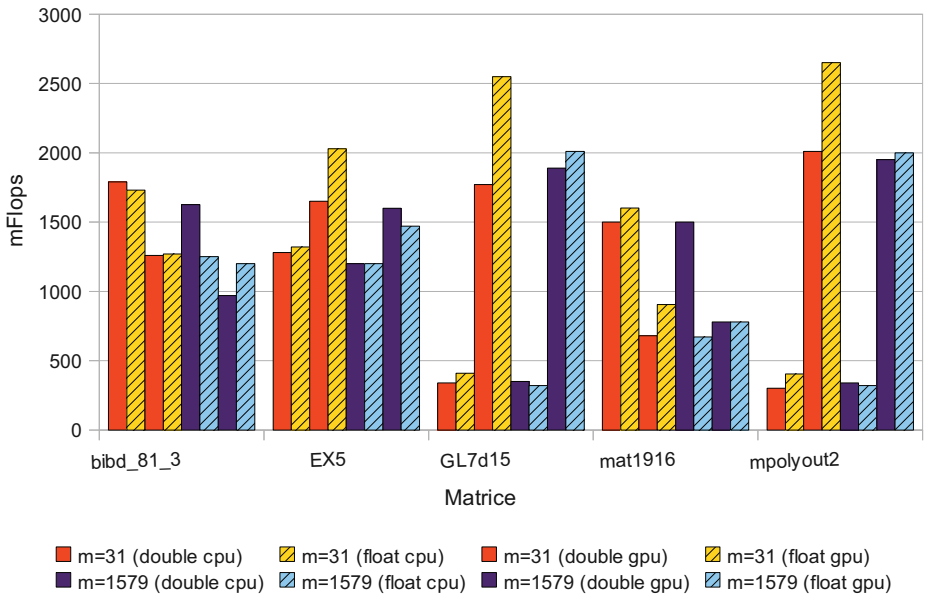

Figure 1: `float`–`double` trade-off for different sizes of m, on the CPU and GPU

In figure 1, we present variations in efficiency due to the storage data type and the size of m on one core of a 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU. The timings correspond to the average of 50 SpMV operations, where $\mathbf{x}$ and $\mathbf{y}$ are randomly generated on the CPU (which also takes into account every data transfer between the CPU and GPU). The measures corresponds to the number of million floating point operations per seconds (flops); a SpMV operation requires $2 * \mathtt{nbnz}$ such operations. The performance correspond to the best ones achieved on the matrices[3] presented in table 1.

name	mat1916	bibd_81_3	EX5	GL7d15	mpolyout2
`row`	1916	3240	6545	460261	2410560
`col`	1916	85320	6545	171375	2086560
`nbnz`	195985	255960	295680	6080381	15707520
rank	1916	3240	4740	132043	1352011

Table 1: Matrices overview

For further information about these techniques on these rings and fast arithmetic in Galois extensions, see e.g. [7].

[3] matrices available at `http://www-ljk.imag.fr/membres/Jean-Guillaume.Dumas/simc.html`

2.3 Adapting numerical libraries

Another speed-up consists in using existing numerical libraries. The ideas behind using them on the rings $\mathbf{Z}/m\mathbf{Z}$, is twofold. Firstly, we delay the modular reduction, secondly we can use highly optimized popular libraries and get instant speed-ups as compared to more naive self-written routines.

Just like BLAS libraries can be used to speed up modular linear algebra [9], we can use numerical libraries for our purposes, or get inspiration for our algorithms from their techniques. For instance, there is the OSKI library [19] for sequential numerical SPMV, or the GPU implementation of SPMV by Nathan Bell *et al.* in [1]. The BLAS specifications include sparse BLAS[4] but they are seldom fully implemented on free BLAS implementations.

Unfortunately, they usually cannot be used as-is. We need to extract submatrices from the sparse matrices, which is more complicated than for its dense counterpart when the use of strides and dimensions suffices. For instance, if one can do b accumulations on $\mathbf{y}[i]$ before reducing and the line i of A has r_i non zero elements. Then we want to split this line between $\lceil r_i/b \rceil$ matrices. Finally, we can use the numerical libraries on these submatrices we have created. The general algorithm reads like follows:

```
spmv(y,A,x){
   foreach submatrix Ai in A do{
      spmv_num(y,Ai,x);
      reduce(y,m);
   }
}
```

Figure 2: Using numerical routines

2.4 New formats

Most of the formats implemented show a row-level parallelism, except `COO` that has element-wise parallelism. The `COO` case is not obvious to implement and generally much slower. The parallel efficiency of other formats will depend then on the length of the rows as well as the data regularity. Unbalanced rows on a GPU architecture will produce many idle threads. Two solutions exist: the vector approach of Bell (they split the rows into shorter chunks) or the rearranging of rows with permutations to sort the row according to their length. The latter will not work in e.g. a power distribution of the row lengths. The `ELL` format answers very well this problem because each row has the same length.

An other way to parallelize the SPMV operation is to split the matrix A along rows to get smaller submatrices and treat them in parallel. We took this approach on the CPU `COO` algorithm.

[4] `www.netlib.org/blas/blast-forum/chapter3.pdf`

Also, we are dealing with large matrices, used many times as black-boxes. Therefore there is a trade-off between the time spent on optimizing the matrix and how much faster these optimizations will make SpMV run.

Things to consider during preprocessing may include for instance: reordering row-columns to create denser parts, choosing best-fitting formats, cutting the matrix into efficient sub-matrices ([20],[16])... The preprocessing approach is taken by Oski: if the expected number of SpMV is very high, optimizing the matrix deeper will prove efficient.

2.4.1 Base case: JIT

One idea to improve SpMV on a given matrix is to hard code this operation in a static library. We read the matrix file and create a library that will apply this matrix to input vectors. For instance the $\mathbf{y} \leftarrow \mathbf{y} + A\,\mathbf{x}$ operation on the matrix $\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$ would be translated to (if $m = 27$)

```
void spmv(float * y, const float * x) {
   y[0] += 2*x[0] ;
   y[0] += x[1] ;
   y[0] = fmod(y[0],27);
   y[1] += 3*x[1] ;
   y[1] = fmod(y[1],27);
}
```

Then we compile this generated file a as static library and use `dlopen` to access its functions. As we can see in this example, one can implement various optimizations: rearranging the rows so that the work is more even (non implemented yet), replacing the occurrences of ± 1 in the matrix by less costly additions or subtractions. We have better control on what the compiler will produce. However, large matrices take extremely long to compile. `gcc` cannot compile the library if the source holds in one huge file, so we divide the matrix into parts of 1000 non zero elements and compile them. Only then for instance, we could compile `bibd_81_3` but it takes 63s on the same Xeon machine. Once it is compiled, the CPU version runs at 620 Mflops, which is reasonably fast.

2.4.2 Taking into account the ± 1

The example of JIT and the observation that many matrices arising from different applications have a lot of $\pm 1_F$ tends to draw our attention to this special case. Moreover, many matrices on a small fields also share this property. Thus we can extract two submatrices corresponding to the 1 and -1 from the rest of the matrix and replace multiplications by usually less expensive additions. Besides, the `data` field in most formats (except `ELL`, `DIA`) can be forgotten as we know they only consist of 1 or -1: this reduces the memory usage. Doing only additions as opposed to `axpy` also hugely delays reduction.

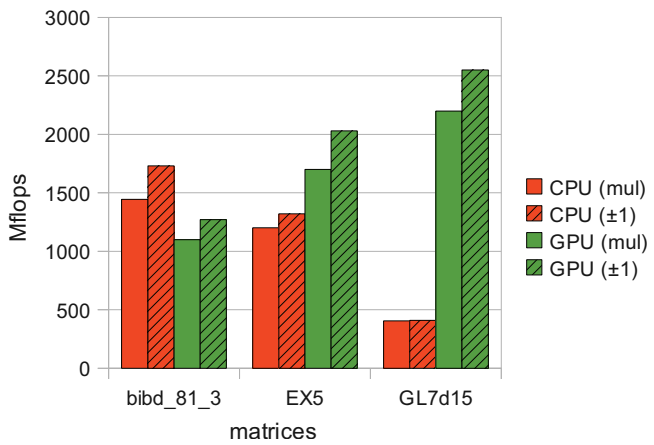

Figure 3: Speed improvement on one 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU when segregating or not the ± 1

Figure 3 shows a maximum 20% improvement on a matrix with only 1s and 15% on matrix with 50% of ± 1.

2.4.3 Basic Formats

As evoked earlier, the matrix A can be split into smaller submatrices. These submatrices can have a format adapted to them and/or can be treated differently. For instance, we can split row-wise and distribute these matrices for parallelism, or split them column-wise as in the delaying case (figure 2. This makes (possibly) many matrices that we each want to optimize individually so we get better overall performance.

We start with some observations. The `COO` format is slow due to the many `fmod` calls, it is best used when the matrix is extremely sparse. The `CSR` format is denser and can let delayed reduction occur, but one has to ensure the row lengths are well balanced when parallelizing. The `ELL` formats are very efficient on matrices that have roughly the same number of non zeros per line. The `ELL_R` format ([17]) is better for uneven rows lengths. One difference in the CPU and GPU architecture makes the `ELL` row-major on the CPU (for better cache use) and column-major on the GPU (for better coalescing). The following figure shows on one example (`bibd_81_3`) the variation of efficiency. The data is normalized so that `CSR` is 1 on the CPU or GPU.

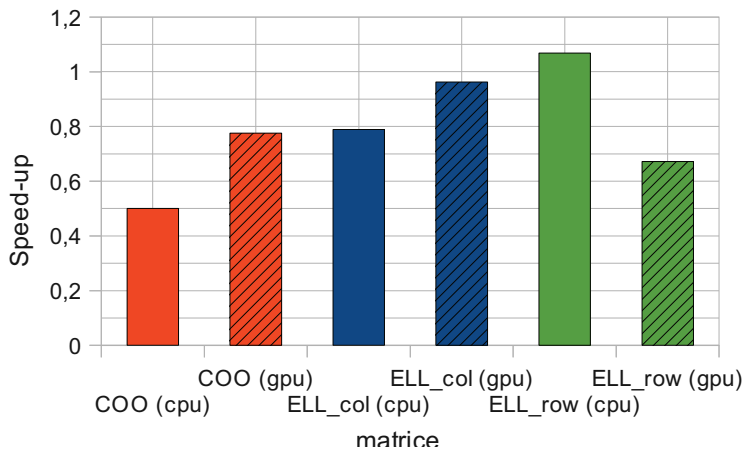

Figure 4: Speed-ups for various formats on matrix `bibd_81_3` both on one 3.2GHz Intel Xeon CPU and a Nvidia GTX280 GPU; reference is `CSR` on each architecture

2.4.4 Hybridization

The previous remarks lead us to combine these formats to take advantage of them. A hybrid format such as `ELL(R)+COO` or `ELL(R)+CRS` leads to good performance on the GPU. When the `ELL` part is taken out of a matrix, many rows can be left empty. Then, we use a format called `COO_S` that is a `CSR` format with pointers only to the non empty rows. It has `data`, `colid` same as in `CSR` and `COO`. The number `rowid`$[k]$ corresponds to the kth non empty row that starts in `data` and `colid` at `start`$[k]$. This format could be avoided if we used row permutations and ordered the lines according to their weight.

2.4.5 Heuristic format chooser

The previous remarks show a great complexity in the formats and the cutting of the matrix. We have implemented a user-helped heuristic format chooser. For instance, the user can indicate if she wants to try and make use of ± 1. If so, for each submatrix, the program tries to find an *a priori* efficient format for them or if it fails, does not separate the 1 or the -1 from the rest. She can also indicate what is the format she wants to fill in priority.

The hybridization of the matrix is usually done as follows. If the matrix is large enough and most of the lines are filled, it will try to fit a part of the matrix in an `ELL` or `ELL_R` format. This choice is supported by the observation that many matrices have a $c + r$ row distribution where c is some constant and $r \in \mathbf{Z}$ varies and the fact that `ELL` is generally much faster that other formats for matrices with even row weight. The rest of the matrix will be put in a `CSR`, `COO` or `COO_S` format, according to the number of empty lines and the number of residual non zero elements. Parameters that decide when segregating

the 1s, that choose the best length for ELL matrix, etc., vary according to the architecture of the computer and need some specific tuning. This tuning is not yet provided at compile time but some of it could be automatically performed at install time.

Experiments show that this heuristic often gives equal or better results that simple formats on the CPU and the GPU.

2.5 Block and iterative versions

2.5.1 Using multi-vectors

We have described the SpMV operation $\mathbf{y} \leftarrow A\,\mathbf{x}$ where $\mathbf{x}$ and $\mathbf{y}$ are vectors. We also need $\mathbf{x}$ and $\mathbf{y}$ to be multi-vectors, for they may be used for block algorithms. There are at least two ways of representing them : row or column-major order. In the row-major order, we can use the standard SpMV many times (and align the vectors). In the column-major order, we can write dedicated versions that try and make use of the cache. Indeed, in this case, we traverse the matrix only once and $\mathbf{x}$ and $\mathbf{y}$ are read/written contiguously.

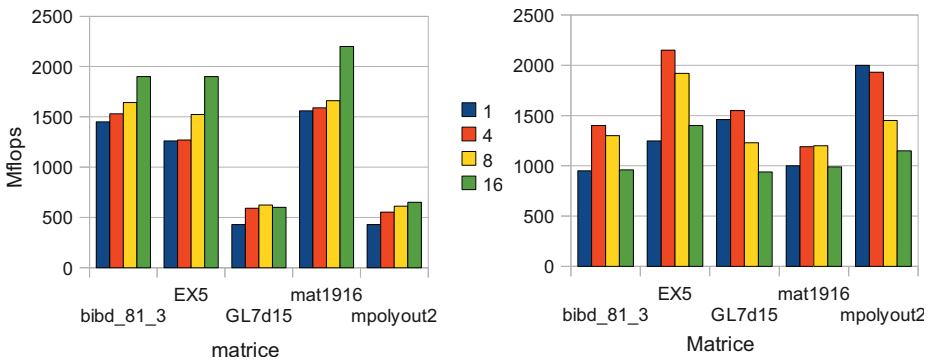

Figure 5: Matrix-multivector multiplication speed on one 3.2GHz Intel Xeon CPU (left) and a Nvidia GTX280 GPU (right) for column-major multi-vectors, with $1, 4, 8$ and 16 vectors.

On figure 5, we note that on the CPU, using column-major multivectors is a non negligible gain of speed. On the contrary, the GPU implementation fails to sustain good efficiency for blocks of more than 8 vectors and some large matrices start to reach the memory limit.

2.5.2 Performance issues

The GPU operation on a single SpMV call from the host point of view is very slow because we need to move the vectors between the host and the device. It is therefore only usable on operations that need no data moving between the host and the device. Examples include the computation $y \leftarrow A^n\ x$ or the

computation of the sequence $\{A^i x\}_{i \in [\![0,m]\!]}$ that are used in many of the black box methods.

On figure 6, we illustrate this differences, mostly reusing or not the data on the GPU, by comparing the performance of the following two pseudo-codes:

```
void smpv_n(y,A,x,n){
  y_d = copy_on_gpu(y);
  x_d = copy_on_gpu(x);
  A_d = copy_on_gpu(A);
  for (i=0 ; i<n ;++i) {
    y_d = A_d * x_d ; // spmv on GPU
    x_d = y_d;        // copy
  }
}
```

```
void n_spmv(y,A,x,n){
  A_d = copy_on_gpu(A);
  for (i=0 ; i<n ;++i) {
    y_d = copy_on_gpu(y_i);
    x_d = copy_on_gpu(x_i);
    y_d = A_d * x_d ; // spmv on GPU
  }
}
```

We confirm on figure 6 that it is highly desirable to not move data on the GPU when avoidable.

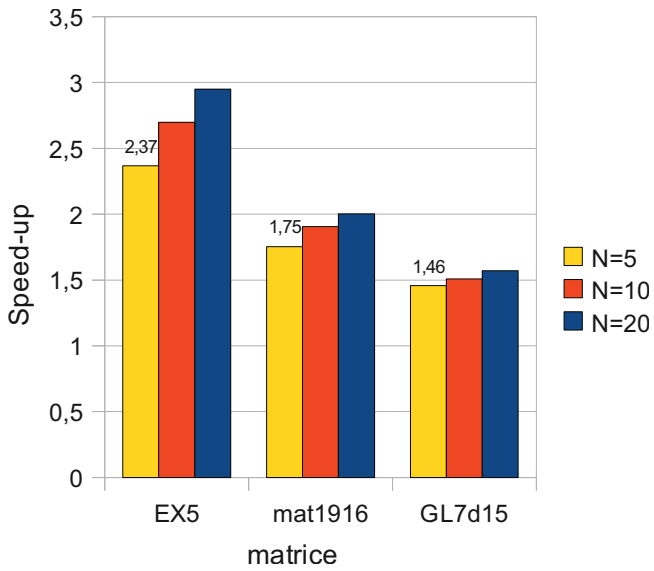

Figure 6: Nvidia GTX280 GPU speed up of $y \leftarrow A^n x$ compared to n times $y \leftarrow Ax$, with $n = 5, 10, 20$

3 Parallel block Wiedemann algorithm

Some of the most representative applications requiring efficient sparse matrix-vector product are blackbox methods based on the Lanczos/Krylov approach. In particular, the method proposed by Wiedemann [21] and its block version proposed by Coppersmith [5] are well suited to highlight efficiency of sparse matrix-vector product since the latter is quite often their bottleneck.

As an application, we propose to improve the implementation of the Block Wiedemann rank algorithm presented in [8]. Let us first briefly recall the outline of this algorithm, we let the reader refer to e.g. [15] for further details.

Let $A \in \mathsf{F}^{n\times n}$ be a matrix satisfying the preconditions of [14]. Then the algorithm can be decomposed in three steps:

1. Compute the matrix sequence $S_i = Y^T A^i Y$ for $i = 0..2n/s + O(1)$, with $Y \in \mathsf{F}^{n\times s}$ chosen at random
2. Compute the minimal matrix generator $F_Y^A \in \mathsf{F}^{s\times s}[x]$ of the matrix series $S(x) = \sum_i S_i x^i$
3. Return the rank $r = \deg(\det(F_Y^A)) - \operatorname{codeg}\det(\mathrm{F}_\mathrm{Y}^\mathrm{A})$.

Our approach is to separate the parallelization of each step. The first step is clearly related to sparse matrix-vector product and we will re-use our tools presented in previous sections. The second step needs the computation of a minimal matrix generator. This can be achieved by a σ-basis computation as explained in [8, section 2.2]. Finally, the last step reduces to computing the co-degree of the determinant of the σ-basis . The degree of the determinant being directly computed as the sum of the row degrees of F_Y^A since, due to the σ-basis properties, the matrix is already in Popov form.

3.1 Parallelization of the matrix sequence generation

The parallelization proposed in [8] was to ship independent set of vector blocks of V to different cores and apply them in parallel. Then gather the results to compute the dense dot products by U^T.

An alternative is to use the SpMV library and let it take care of the iteration with the algorithm of the preceding section.

In figure 7 we compare both approaches:

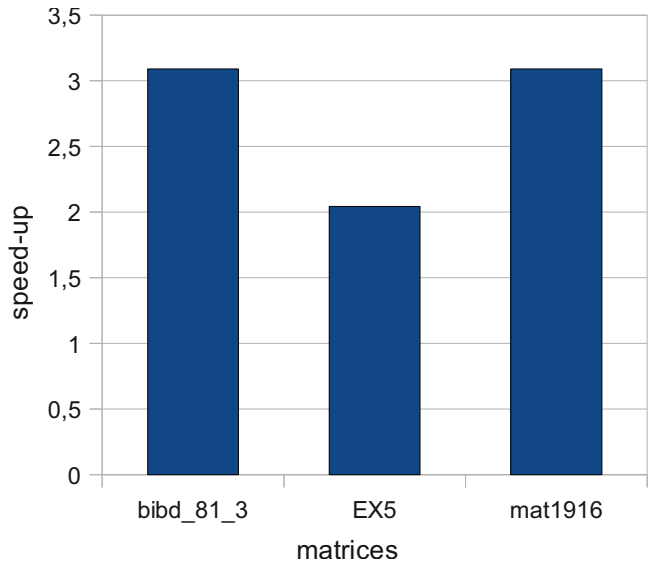

Figure 7: Speed up from the new SpMV library compared to the native LinBox implementation in the generation of the matrix sequence ($2n$ iterations) on one core of a 2.33GHz Intel Xeon E5345 CPU

3.2 Parallelization of the σ-basis computation

One can efficiently compute σ-basis using the algorithm PM-Basis of [11]. This algorithm mainly reduces to polynomial matrix multiplication. Therefore a first parallelization approach is to parallelize the polynomial multiplication.

3.2.1 Parallel polynomial matrix multiplication

Let $A, B \in \mathsf{F}^{n\times n}[x]$ be two polynomial matrices of degree d. One can multiply A and B in $O(n^3 d + n^2 d \log d)$ operations in F assuming F has a d-th primitive root of unity [3]. Assuming one has k processors such that $k \leqslant n^2$, one can perform this multiplication with a parallel complexity of $O(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k})$ operation in F. Let us now see the sequential fast polynomial matrix multiplication algorithm and how it achieves such a parallel complexity:

Fast Polynomial Matrix Multiplication:
Inputs: $A, B \in \mathsf{F}^{n\times n}[x]$ of degree d, ω a d-th primitive root of unity inf F.
Outputs: $A \times B$

1. $\bar{A} := DFT(A, [1, \omega, \omega^2, ..., \omega^{2d}])$
2. $\bar{B} := DFT(B, [1, \omega, \omega^2, ..., \omega^{2d}])$
3. $\bar{C} := \bar{A} \otimes \bar{B}$
4. $C := \frac{1}{2d} DFT(\bar{C}, [1, \omega^{-1}, \omega^{-2}, ..., \omega^{-2d}])$

return C.

Here, $DFT(P, L)$ means the multi-points evaluation of the polynomial P on each points of L, while $\otimes$ means the point-wise product.

- step 1,2 and 4 can be accomplished by using Fast Fourier Transform on each matrix entries which gives $n^2 \times O(d \log d)$ operations (see [10, Theorem 8.15]). This clearly can be distributed on k processors such that each processor achieves in parallel the FFT on $\frac{n^2}{k} + O(1)$ matrix entries. This gives a parallel complexity of $O(\frac{n^2 d \log d}{k})$ operations in F.
- step 3 requires the computation of $2d$ independent matrix multiplications of dimension n, which gives $O(n^3 d)$ operations in F. One can easily see how to distribute this work on k processors such that each processor has a workload of $O(\frac{n^3 d}{k})$ operations.

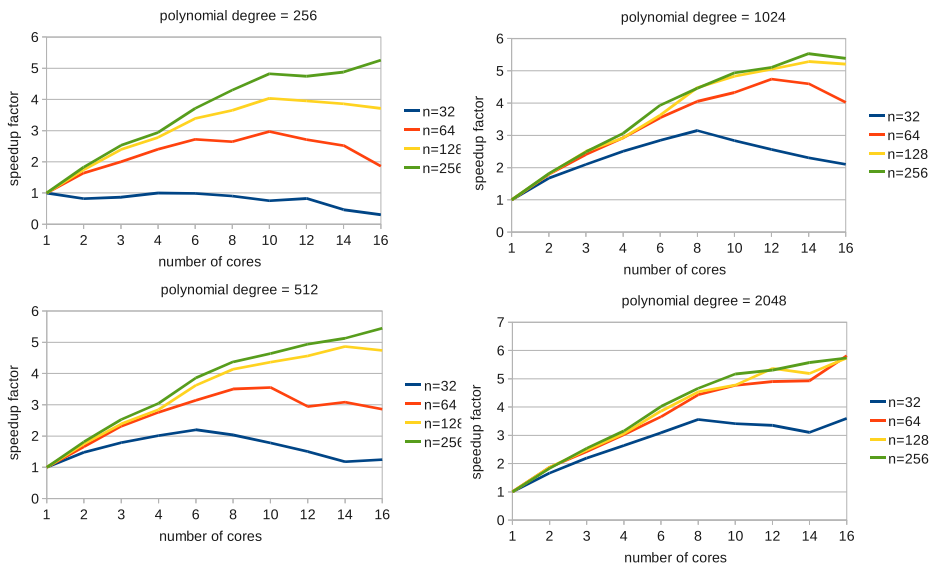

Figure 8: Scalability of parallel polynomial matrix multiplication with LinBox and OpenMP on a 16 core machine (based on Quad-Core AMD Opteron). n is the matrix dimension.

We report in figure 8 the performance of the implementation of this parallel algorithm in the LinBox[5] library. Our choice of using this parallel algorithm rather than another one, achieving a possible better parallel complexity, has been driven by the re-usability of efficient sequential components of the library (e.g. matrix multiplication) and the ease of use within the library itself (i.e. mostly the same code as sequential one, only some OpenMP pragmas have been

[5] www.linalg.org

added).

One can see on figure 8 that our coder does not completely match the theoretical parallel speedup. The best we can achieved with 16 processors is a speedup of 5.5, which is only one third of the theoretical optimality. Nevertheless, one can see that with less processors (e.g. less than 4) the speedup factor is closer to 75% of the optimality, which is quite fair. We think this phenomenon can be explained by the underlying many multi-core architecture (Quad-Core AMD Opteron), which may clearly suffers from cache effect if computation are done on same chip or not.

As expected, we can also point out from figure 8 that our implementation benefits at most from parallelism when matrices are larger. Since workload on each core is more important, this allows to hide the penalty from memory operations and threads management of OpenMP. This remarks also applies on the degree but the impact is less important.

3.2.2 Parallel σ-basis implementation

According to the reduction of PM-Basis to polynomial matrix multiplication, one can achieve a parallel complexity of $\tilde{O}(\frac{n^3 d}{k} + \frac{n^2 d \log d}{k})$ operations in F with k processors for σ-basis calculation, assuming $k \leqslant n^2$. Therefore, it suffices to directly plug in our parallel polynomial matrix multiplication into the original code of the LinBox library to get a parallel σ-basis implementation.

We report in figure 9 the performance of the parallel version of PM-Basis algorithm within LinBox. Here again, the speedup factor of parallelism is quite low when compared to the theoretical optimality. At most we were able to obtain a speedup of 3 with 16 processors. However, this timings are consistent with the previous ones in figure 8 where the best speedup was 5.

One may notice that reduction to polynomial matrix multiplication of the PM-Basis algorithm relies on a divide a conquer approach on the degree of the approximation (see [11, theorem 2.4]). Therefore, the recursion calls are made with smaller and smaller approximation's degrees, which leads to use less efficient parallel multiplications. Moreover, when the degree is too small the use of the M-Basis algorithm of [11] should be prefered since it becomes more efficient in practice. We have not yet implemented a parallel implementation of this algorithm in LinBox and this clearly affects the performance of our implementation.

3.3 Parallel determinant co-degree

Here we just launch in parallel the evaluations of the matrix polynomial at different points, and the computation of the determinant of the obtained matrix at the given point, and gather the results sequentially with the `Poly1CRT` class of Givaro.

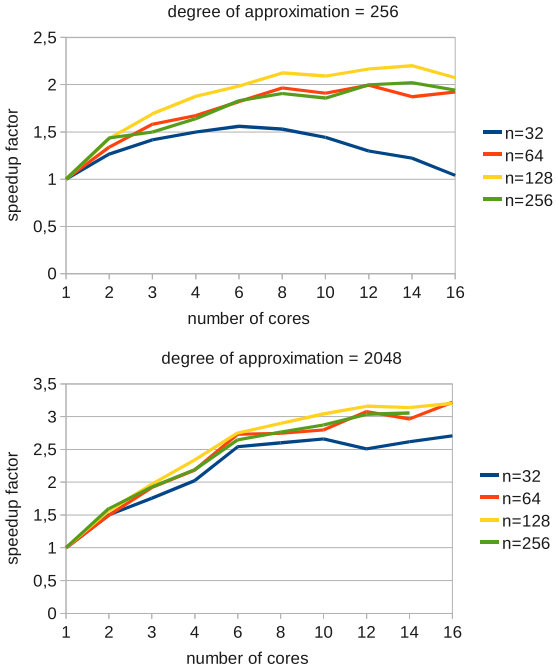

Figure 9: Scalability of parallel σ-basis computation with LinBox and OpenMP on a 16 core machine (based on a Quad-Core AMD Opteron). n is the matrix dimension of the series.

3.4 Parallel block Wiedemann performance

In table 2 we show the overall performance of our algorithm on an octo-processor Xeon E5345 CPU, 8×2.33GHz. `*-LB` shows the timings of the current LinBox implementation, where `*-SpMV` presents our new improvement, both in sequential and in parallel. The speed-up for SpMV between 1 and 8 processors is slightly larger than 5 for all the matrices where the speed-up for LinBox ranges from 4 to 4.9. Furthermore, the speed-up obtained with SpMV versus LinBox on the sequence generation seems scalable as it even improves when used in a parallel setting.

4 Conclusion

We have proposed a new SpMV library providing good results on $\mathbf{Z}/m\mathbf{Z}$ rings. To attain this efficiency it has been mandatory to augment the complexity of the SpMV algorithms, since OpenMP, Cuda et al. all manage differently the parallelization. Nonetheless, we provide new hybrid formats that improve the performance. Moreover we have also specialized it to the computation of a sequence of matrix-vector products together with a new parallelization of the

Matrix	mat1916		bibd_81_3		EX5	
Cores	1	8	1	8	1	8
Seq-LB	15.09	3.08	47.73	12.41	84.21	20.22
Seq-SpMV	5.02	0.91	41.28	7.56	49.66	7.36
σ-basis	9.02	1.64	18.45	3.63	37.45	8.39
Interpolation	0.37	0.29	1.07	0.82	2.29	1.75
Total-LB	24.48	5.01	67.25	16.86	123.95	30.36
Total-SpMV	14.41	**2.84**	60.80	**12.01**	89.40	**17.50**

Table 2: Rank modulo 65521 with OpenMP Parallel block Wiedemann on a Xeon E5345, 8 × 2.33GHz

sigma-basis algorithm in order to enhance e.g. rank computations of very large sparse matrices.

As seen in 3.2.2, a first parallelization of the σ-basis computation has been achieved. Its efficiency is not matching the expected scalability and lot of work needs to be done to circumvent this problem. First, a deeper study on the parallelization of σ-basis computation has to be done. Beside the parallelization of PM-Basis and M-Basis algorithms themselves, we need to design new algorithms to avoid the numerous task dependencies, inherent to the existing methods. This will also enable an easier parallelization of early termination strategies (requiring to interleave the generation sequence and the σ-basis computation).

Another important task is to extend the sigma-basis algorithm to work on polynomial matrices over extension fields. Indeed the use of random projections U and V over extension fields might improve the probabilities to get the full minimal polynomial of the matrix [12, 18, 4]. As shown in this paper and in [8], σ-basis needs only a polynomial matrix multiplication implementation to work. In order to adapt current LinBox's implementation to extension field, we will use the same technique as [7]: first use Kronecker substitution to transform the extension field polynomial representation to an integer representation ; then use a Chinese remaindered version of the polynomial matrix multiplication to recover the resulting matrix polynomial over Z ; and finally convert back the integers using e.g. the REDQ inverse operation of [6].

The SpMV implementation also needs further work and other directions to be explored. For instance, we need to have dedicated implementations in $\mathbf{Z}/2\mathbf{Z}$ where $\mathbf{x}$ and $\mathbf{y}$ can be compressed. More formats, including dense submatrices, have yet to be explored, which is linked to spending some more time on pre-processing the matrix: for instance the use of Metis[6] for partitioning and reordering A would also improve the performance. It will be interesting to deal with matrices such that A and ${}^{\top}A$ cannot be simultaneously stored ([2]). This problem indeed occurs on GPU's where on-chip memory is very limited. Finally, we will also provide multi-GPU and hybrid GPU/CPU implementations.

[6] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

References

[1] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[2] 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 *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, New York, NY, USA, 2009. ACM.

[3] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[4] L. Chen, W. Eberly, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and its Applications*, 343-344:119–146, 2002.

[5] D. Coppersmith. Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. *Mathematics of Computation*, 62(205):333–350, Jan. 1994.

[6] J.-G. Dumas. Q-adic transform revisited. In D. Jeffrey, editor, *Proceedings of the 2008 ACM International Symposium on Symbolic and Algebraic Computation, Hagenberg, Austria*, pages 63–69. ACM Press, New York, July 2008.

[7] J. G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 63–74, New York, NY, USA, 2002. ACM.

[8] J.-G. Dumas, P. Giorgi, P. Elbaz-Vincent, and A. Urbańska. Parallel computation of the rank of large sparse matrices from algebraic k-theory. In S. Watt, editor, *PASCO 2007*, pages 43–52. Waterloo University, Ontario, Canada, July 2007.

[9] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

[10] J. v. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[11] P. Giorgi, C.-P. Jeannerod, and G. Villard. On the complexity of polynomial matrix computations. In R. Sendra, editor, *Proceedings of the 2003 ACM International Symposium on Symbolic and Algebraic Computation,*

Philadelphia, Pennsylvania, USA, pages 135–142. ACM Press, New York, Aug. 2003.

[12] E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, Apr. 1995.

[13] E. Kaltofen and A. Lobo. Factoring high-degree polynomials by the black box Berlekamp algorithm. In ACM, editor, *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation: July 20–22, 1994, Oxford, England, United Kingdom*, pages 90–98, pub-ACM:adr, 1994. ACM Press.

[14] E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In *Applied Algebra, Algebraic Algorithms and Error–Correcting Codes (AAECC '91)*, volume 539 of *Lecture Notes in Computer Science*, pages 29–38, Oct. 1991.

[15] W. J. Turner. A block Wiedemann rank algorithm. In J.-G. Dumas, editor, *Proceedings of the 2006 ACM International Symposium on Symbolic and Algebraic Computation, Genova, Italy*, pages 332–339. ACM Press, New York, July 2006.

[16] P. Tvrdik and I. Simecek. A new approach for accelerating the sparse matrix-vector multiplication. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on*, 0:156–163, 2006.

[17] F. Vazquez, E. M. Garzon, J. A. Martinez, and J. J. Fernandez. The sparse matrix vector product on GPUs. *Technical Report*, June 2009.

[18] G. Villard. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Technical Report 975–IM, LMC/IMAG, Apr. 1997.

[19] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.

[20] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In L. T. Yang, O. F. Rana, B. D. Martino, and J. Dongarra, editors, *HPCC*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816, pub-SV:adr, 2005. Springer-Verlag Inc.

[21] D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, Jan. 1986.