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

Brice Boyer  Jean-Guillaume Dumas*
Laboratoire J. Kuntzmann, UMR CNRS 5224, Université de Grenoble
BP 53X, F38041 Grenoble, France.
{Brice.Boyer,Jean-Guillaume.Dumas}@imag.fr

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

ABSTRACT
We propose different implementations of the sparse matrix–
dense vector multiplication (SpMV) for finite fields and rings
\( \mathbb{Z}/m\mathbb{Z} \). 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 li-
brary and a new parallelisation of the sigma-basis algorithm
in a parallel block Wiedemann rank implementation over fi-
nite fields.

Categories and Subject Descriptors
F.2.1 [Numerical Algorithms and Problems]: Computations in finite fields; G.1.3 [Numerical Linear Algebra]: Sparse, structured, and very large systems (direct and iterative methods); G.4 [Mathematical Software]: Parallel and vector implementations

Keywords
Sparse Matrix Vector multiplication, Finite field, Parallelism, Rank, Block Wiedemann

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

*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.

On the numerical side, several libraries automatically tune
the sparse matrix kernels [19, 20, 16] and recently some ker-
nels have been proposed for GPU’s [17, 2, 1]. In this paper
we want to adapt those techniques for exact computations
so we first focus on \( \mathbb{Z}/m\mathbb{Z} \) rings, with \( m \) smaller that a
machine word.

The first idea consists in using the numerical methods in
an exact way as done for dense matrix operations [7].
For sparse matrices, however, the extraction of sparse sub-
matrices is different. Also, over small fields some more dedi-
cated 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 dynami-
cally using ad-hoc sparse matrix completions. The idea is
that a given matrix will have different parts in different for-
mat formats adapted to its data or to the resources. Also we outline
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 multi-
plication” (SpMV) on finite rings, together with the trans-
pose product and the iterative process to compute the power
of a matrix times a vector.

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 multipli-
cation 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 com-
pute minimal polynomials [11, 8].

In section 2 we present different approaches to the par-
allelization of the SpMV operation: the adaptation of nu-
merical libraries (section 2.3) and new formats adapted to
small finite rings (section 2.5) together with our new hy-
brid strategy and their iterative versions (section 2.6). Then
in section 3 we propose a new parallelization of the block
Wiedemann rank algorithm in LinBox, via the paralleliza-
tion of the matrix-sequence generation (section 3.1) and the
parallelization of the matrix minimal polynomial computa-
tion (section 3.2).

\(^1\)https://ljkforge.imag.fr/projects/ffspmvgpu/
\(^2\)http://linalg.org

Permission to make digital or hard copies of all or part of this work
for personal or classroom use is granted without fee provided that
copies are not made or distributed for profit or commercial
advantage and that copies bear this notice and the full citation
on the first page. To copy otherwise, to
republish, to post on servers or to redistribute to lists, requires prior
specific permission and/or a fee.

Copyright 2010 ACM 978-1-4503-0067-4/10/0007 ..$10.00.
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-zero elements in matrix $A$, it has row lines and col columns. If $x$ and $y$ are vectors, then we perform here the operation $y ← A\times x + y$. The general operation $y ← αA\times x + βy$ can then be done by pre-multiplying $x$ and $y$ by $α$ and $β$ respectively. The operation $y ← A^\top x + y$ is done by pre-transposing the matrix $A$. The apply operation in LinBox black box algorithms, or $y ← A\times x$, is performed by first setting the elements of $y$ to zero. For further use in block methods, we also provide the operation $Y ← αAX + βY$ where $X$ and $Y$ are sets of vectors (or multivectors).

2.1 Sparse Matrix Formats and Multiplication

Sparse matrices arise from various domains and their shapes can be very specific. Taking into consideration the structure of a sparse matrix can dramatically 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 $\text{data}$, $\text{colid}$ and $\text{rowid}$, such that $\text{data}[k] = A[\text{rowid}[k], \text{colid}[k]]$.

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 $\text{start}[i] ≤ k < \text{start}[i+1]$, then $\text{data}[k] = A[i,\text{colid}[k]]$. 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 $\text{data}$ and $\text{colid}$ fields such that $\text{data}[i, j0] = A[i, \text{colid}[i,j0]]$, where $j0$ 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 rrownb 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, first taking advantage of this variety of formats.

2.2 Finite field representation

We present now how the data is stored. We use common data types such as float, int... Firstly, when doing modular linear algebra over $\mathbb{Z}/m\mathbb{Z}$, we try to minimize the number of costly $\text{fmod}$ (reduction by mod $m$) operation calls. Indeed, this $\text{fmod}$ operation is the one difference with numerical implementations of SpMV and needs very special attention. For instance, we prefer, if possible, the right loop to the left one in the following figure 1:

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

Figure 1: Delaying fmod

In this case, suppose that $y = 0$, $a[i]$ and $b[i]$ are reduced modulo $m$ at first. Let $M$ be the largest representable integer. On $\mathbb{Z}/m\mathbb{Z}$, we can represent the ring on $[0,m-1]$. Then we can do at most $M/(m-1)^2$ accumulations before reducing. We can also represent the ring on $[-\left\lfloor \frac{m-1}{2}\right\rfloor, \left\lceil \frac{m-1}{2}\right\rceil]$. This last 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.

In figure 2, 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 $x$ and $y$ are randomly generated on the CPU; every data transfer between the CPU and GPU is taken into account. The measure unit corresponds to the number of million floating point operations per seconds (flops); a SpMV operation requires $2*nbnz$ such operations. The data storage here is ELL_R and the matrices\(^4\) are presented in table 1. This figure shows a significant slow down for for double operations on this GPU and a minor one on this CPU. On the border case (large prime, small data type), we can see a large performance gain on the CPU but not on the GPU. We suspect the massive parallelisation on the GPU hides the cost of the fmod operation.

2.3 Adapting numerical libraries

Another speed-up consists in using existing numerical libraries. The ideas behind using them on the rings $\mathbb{Z}/m\mathbb{Z}$ 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 naïve “home made” 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\footnote{www.netlib.org/blas/blast-forum/chapter3.pdf} but these routines are seldom fully implemented in free BLAS implementations.

Unfortunately, numerical libraries 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, let us suppose one can do $b$ accumulations on $y[i]$ before an overflow may happen and we need to reduce. Suppose too that line $i$ of $A$ has $r_i$ non zero elements. Then we want to split this line between $[r_i/b]$ matrices. We can improve this technique with a finer majoration. We split the elements in row $i$ into a disjoint union of $k_i$ sets $S_{i,k}$. Let $\mu$ be the largest (in absolute value) element we can represent in the ring (usually $m$ or $[m/2]$). For all $i,k$, we demand that $\sum_{j \in S_{i,k}} |a_{i,j}| \mu < M$ and create $\max(k_i)$ submatrices.

Eventually, we can use the numerical libraries on these submatrices we have created. The general algorithm reads as follows:

```c
spmv(float * y, const float * x) {
    foreach submatrix $A_i$ in $A$ do {
        spmv_num(y,$A_i$,x); //no overflow guaranteed
        reduce(y,x);}
}
```

2.4 Using OpenMP

For the ease of implementation and for it being a well-used standard, we chose OpenMP to parallelise our code on the CPU. In most cases, we simply added a #pragma omp above the outer loops. This technique gave good performances as shown in figure 4.

As we can notice on this figure, OpenMP help scale well the performances, with a few processors.

2.5 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 is generally much slower. The parallel efficiency of other formats will depend 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 et al. (they split the rows into shorter chunks and reduce) or the rearranging of rows with permutations to sort the rows according to their length. The last idea 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. However, one has to be very careful about the amount of unnecessary memory allocated by ELL in case of very uneven row lengths. Then, as proved in e.g. [1], yet another solution consists in splitting the matrix $A$ into a sum of matrix, one dense part in ELL, the other in COO format.

An other way to parallelise 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.

Also, we have to keep in mind that 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 (\cite{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.

2.5.1 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 $y \leftarrow y + Ax$ operation on the matrix $\begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$ would be translated to (if $m = 27$):

```c
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 as a 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, replacing the occurrences of \pm 1 in the matrix by less costly additions or subtractions. However, large matrices take extremely long to compile, even if the matrix is divided into smaller (easier to compile) submatrices. Only then for instance, we could compile `bibd_81_3` but it took 63s on the same Xeon machine. Once it is compiled, the CPU version runs at 620 Mflops, which is reasonably fast but not usable. This idea did not take into account the pressure on the instruction cache and the possibly small bandwidth of certain architectures. However, it produced the idea for the following data formats.

### 2.5.2 Taking into account the \pm 1

The example of JIT and the observation that many matrices arising from different applications have a lot of \pm 1 attracted our attention on 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.

### 2.5.3 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 3). 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 less even row 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.

![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.](image)

**Figure 6**: 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.

### 2.5.4 Hybridization

The previous remarks lead us to combine these formats to take advantage of them. Hybrid formats such as `ELL(_R)+COO` or `ELL(_R)+CRS` give 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 `rovid[k]` corresponds to the \(k^{th}\) 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.

### 2.5.5 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 \pm 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 \(-1\) 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, on one hand, many matrices have a \(c + r\) row distribution where \(c\) is some constant and \(r \in \mathbb{Z}\) varies and that, on the other hand, `ELL` is generally much faster that other formats for matrices
with even row weights. 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 (figure 7) show that this heuristic often gives equal or better results that simple formats on the CPU and the GPU.

2.6 Block and iterative versions

2.6.1 Using multi-vectors

We have described the SpMV operation $y \leftarrow Ax$ where $x$ and $y$ are vectors. We also need $x$ and $y$ to be multi-vectors, for they may be used in block algorithms. There are at least two ways to represent 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 $x$ and $y$ are read/written contiguously.

![Figure 8: Matrix-multivector multiplication speed on one 3.2GHz Intel Xeon CPU (left) and a Nvidia GTX280 GPU (right) for column-major multivectors, with 1, 4, 8 and 16 vectors. (ELL_R format)](image)

On figure 8, 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. We suspect the problem comes from a bad use of local memory. Besides, some large matrices start to reach the memory limitation.

2.6.2 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 10, we illustrate this difference, mostly reusing or not the data on the GPU, by comparing the performance of the following two pseudo-codes (figure 9):

```c
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; // full 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
    }
}
```

![Figure 9: Pseudo code for $y \leftarrow A^n x$ and $n$ times $y \leftarrow Ax$ on the GPU with $x$ randomly generated on the CPU.](image)

As expected, figure 10 clearly supports that it is highly desirable not to move data between CPU and GPU when avoidable. The speed-up is noticeable even from a very small number of iterations.

![Figure 10: Nvidia GTX280 GPU speed up of $y \leftarrow A^n x$ compared to $n$ times $y \leftarrow Ax$, with $n = 5, 10, 20$; CSR format is used.](image)

Note : plots throughout this paper all take into account the data transfer between CPU/GPU and back at each SpMV operation.

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 $A ∈ F^{n×n}$ 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 $Y ∈ F^{n×s}$ chosen at random
2. Compute the minimal matrix generator $F_{\sigma} \in F^{s×s}[x]$ of the matrix series $S(x) = \sum_i S_i x^i$
3. Return the rank $r = \deg(\det(F_{\sigma})) - \cdet(\det(F_{\sigma}))$

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 $\sigma$-basis computation as explained in [8, section 2.2]. Finally, the last step reduces to computing the co-degree of the determinant of the $\sigma$-basis. The degree of the determinant being directly computed as the sum of the row degrees of $F_{\sigma}$ since, due to the $\sigma$-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 11 we compare both approaches:

![Figure 11: 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](www.linalg.org)

### 3.2 Parallelization of the $\sigma$-basis computation

One can efficiently compute $\sigma$-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 $A, B ∈ F^{n×n}[x]$ be two polynomial matrices of degree $d$. One can multiply $A$ and $B$ in $O(n^3d + n^2d \log d)$ operations in $F$ assuming $F$ has a $d$-th primitive root of unity $\omega$. Assuming one has $k$ processors such that $k ≤ n^2$, one can perform this multiplication with a parallel complexity of $O(\frac{n^3d}{k} + \frac{n^2d \log d}{k})$ 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:** $A, B ∈ F^{n×n}[x]$ of degree $d$, $\omega$ a $d$-th primitive root of unity in $F$.

**Outputs:** $A \times B$

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

return $C$.

Here, $DFT(P, L)$ means the multi-points evaluation of the polynomial $P$ on each points of $L$, while $\otimes$ means the pointwise 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 $O(\frac{n^2d \log d}{k})$ operations in $F$.

- step 3 requires the computation of $2d$ independent matrix multiplications of dimension $n$, which gives $O(n^3d)$ operations in $F$. One can easily see how to distribute this work on $k$ processors such that each processor has a workload of $O(\frac{n^3d}{k})$ operations.

We report in figure 12 the performance of the implementation of this parallel algorithm in the LinBox5 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 added).

One can see on figure 12 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...
As expected, we can also point out from figure 12 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 \( \sigma \)-basis implementation

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

We report in figure 13 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 12 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.

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 x 2.33GHz. \( \ast \)-LB shows the timings of the current LinBox implementation, where \( \ast \)-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. Comparing with figure 11, we can confirm that the bottle-neck in the sequence phase is really the SpMV operation.

This table will be completed in the future with a line including the generation of the sequence on the GPU. It involves porting some FFLAS/FFPACK functionalities to the GPU using CUBLAS which is underway.

4. CONCLUSION

We have proposed a new SpMV library providing good results on \( \mathbb{Z}/m \mathbb{Z} \) 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 sparse-matrix-vector products together with a new parallelization of the sigma-basis algorithm in order to enhance e.g. rank computations of very large sparse matrices. As seen in 3.2.2, a first sigma-basis algorithm itself 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 the sigma-basis computation has been achieved. 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], $\sigma$-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 $\mathbb{Z}$, 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 $\mathbb{Z}/2\mathbb{Z}$ where $x$ and $y$ 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\textsuperscript{6} for partitioning and reordering $A$ would also improve the performance. It will be interesting to deal with matrices such that $A$ and $A^t$ cannot be simultaneously stored (\cite{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.

5. REFERENCES


\textsuperscript{6}http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Matrix & mat1916 & bibleL81_3 & EX5 \\
\hline
\hline
Cores & I & 8 & I & 8 & I & 8 \\
\hline
Seq-LB & 15.09 & 3.08 & 47.73 & 12.41 & 84.21 & 20.22 \\
\hline
Seq-SpMV & 5.02 & 0.91 & 41.28 & 7.56 & 49.66 & 7.36 \\
\hline
$\sigma$-basis & 9.02 & 1.64 & 18.45 & 3.63 & 37.45 & 8.39 \\
\hline
Interpolation & 0.37 & 0.29 & 1.07 & 0.82 & 2.29 & 1.75 \\
\hline
Total-LB & 24.48 & 5.01 & 67.25 & 16.86 & 123.95 & 30.36 \\
\hline
Total-SpMV & 14.41 & 2.84 & 60.80 & 12.01 & 89.40 & 17.50 \\
\hline
\end{tabular}
\caption{Table 2: Rank modulo 65521 with OpenMP Parallel block Wiedemann on a Xeon E5345, 8 × 2.33GHz (timings in seconds)}
\end{table}
algebraic computation, pages 63–74, New York, NY, USA, 2002. ACM.


