Accelerating Iterative SpMV for Discrete Logarithm Problem Using GPUs

Hamza Jeljeli

Accelerating Iterative SpMV for the Discrete Logarithm Problem Using GPUs

Hamza Jeljeli

CARAMEL project-team, LORIA, INRIA / CNRS / Université de Lorraine,
Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
Hamza.Jeljeli@loria.fr

Abstract. In the context of cryptanalysis, computing discrete logarithms in large cyclic groups using index-calculus-based methods, such as the number field sieve or the function field sieve, requires solving large sparse systems of linear equations modulo the group order. Most of the fast algorithms used to solve such systems — e.g., the conjugate gradient or the Lanczos and Wiedemann algorithms — iterate a product of the corresponding sparse matrix with a vector (SpMV). This central operation can be accelerated on GPUs using specific computing models and addressing patterns, which increase the arithmetic intensity while reducing irregular memory accesses. In this work, we investigate the implementation of SpMV kernels on NVIDIA GPUs, for several representations of the sparse matrix in memory. We explore the use of Residue Number System (RNS) arithmetic to accelerate modular operations. We target linear systems arising when attacking the discrete logarithm problem on groups of size 100 to 1000 bits, which includes the relevant range for current cryptanalytic computations. The proposed SpMV implementation contributed to solving the discrete logarithm problem in $\mathrm{GF}(2^{619})$ and $\mathrm{GF}(2^{809})$ using the FFS algorithm.

Keywords: Discrete Logarithm Problem, Sparse-Matrix–Vector product, Modular Arithmetic, Residue Number System, GPUs.

1 Introduction

The security of many cryptographic protocols used for authentication, key exchange, encryption, or signature, depends on the difficulty of solving the discrete logarithm problem (DLP) in a given cyclic group [16]. For instance, we can rely on the hardness of the DLP in a multiplicative subgroup of a finite field. There are algorithms, such as Pollard-rho [17] or Baby-Step/Giant-Step [21] that solve the problem in time exponential in the subgroup size. Another family of methods, known as *Index-calculus* methods [1] propose to solve it in time sub-exponential or quasi-polynomial in the finite field size. These algorithms require in their linear algebra step the resolution of large sparse systems of linear equations modulo the group order [12]. In cryptographic applications, the group order ℓ is of size 100 to 1000 bits. The number of rows and columns of the corresponding matrices is in the order of hundreds of thousands to millions, with only hundreds or fewer

non-zero elements per row. This linear algebra step is a serious limiting factor in such algorithms. For example, it was reported in [9] that the linear algebra step of the Function Field Sieve (FFS) implementation to solve the DLP over GF($3^{6\times97}$) took 80.1 days on 252 CPU cores, which represents 54% of the total time.

To solve such systems, ordinary Gaussian elimination is inefficient. While some elimination strategies aiming at keeping the matrix as sparse as possible can be used to reduce the input system somewhat, actual solving calls for the use of other techniques (Lanczos algorithm [13], Wiedemann algorithm [27]) that take advantage of the sparsity of the matrix [18]. For the Lanczos algorithm, the Wiedemann algorithm and their block variants, the iterative sparse-matrix–vector product is the most time-consuming operation. For this reason, we investigate accelerating this operation on GPUs.

The paper is organized as follows. Section 2 presents the background related to the hardware and the context. Section 3 discusses the arithmetic aspects of our implementation. We present several matrix formats and their corresponding implementations in Sections 4. We discuss in Section 5 how to adapt these implementations over large fields. We compare the results of different implementations run on NVIDIA GPUs in Section 6, and present optimizations based on hardware considerations in Section 7. Section 8 discusses our reference software implementation.

2 Background

2.1 GPUs and the CUDA programming model

CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written in C, C++, OpenCL and other languages [14].

A CUDA program instantiates a *host* code running on the CPU and a *kernel* code running on the GPU. The kernel code runs according to the Single Program Multiple Threads (SPMT) execution model across a set of parallel threads. The threads are executed in groups of 32, called *warps*. If one or more threads have a different execution path, execution divergence occurs. The different paths will then be serialized, negatively impacting the performance.

The threads are further organized into thread *blocks* and *grids* of thread blocks:

- A thread executes an instance of the kernel. It has a unique thread ID within its thread block, along with registers and private memory.
- A thread block is a set of threads executing the same kernel which can share data through *shared memory* and perform barrier synchronization which ensures that all threads within that block reach the same instruction before continuing. It has a unique block ID within its grid.
- A grid is an array of thread blocks executing the same kernel. All the threads of the grid can also read inputs, and write results to *global memory*.

At the hardware level, the blocks are distributed on an array of multi-core *Streaming Multiprocessors* (SMs). Each SM schedules and launches the threads in groups of warps. Recent NVIDIA GPUs of family name "*Kepler*" allow for up

to 64 active warps per SM. The ratio of active warps to the maximum supported is called *occupancy*. Maximizing the occupancy is important, as it helps to hide the memory latency. One should therefore pay attention to the usage of shared memory and registers in order to maximize occupancy.

Another important performance consideration in programming for the CUDA architecture is *coalescing* global memory accesses. To understand this requirement, global memory should be viewed in terms of aligned segments of 32 words of 32 bits each. Memory requests are serviced for one warp at a time. If the warp requests hit exactly one segment, the access is *fully coalesced* and there will be only one memory transaction performed. If the warp accesses scattered locations, the accesses are *uncoalesced* and there will be as many transactions as the number of hit segments. Consequently, a kernel should use a coalescing-friendly pattern for greater memory efficiency.

Despite their high arithmetic intensity and their large memory bandwidth, GPUs provide small caches. In fact, Kepler GPUs provide the following levels of cache:

- 1536-kB *L2-cache* per GPU.
- 16-kB *L1-cache* (per SM). It can be extended to 48 kB, but this decreases shared memory from 48 kB to 16 kB.
- A *texture cache*: an on-chip cache for the read-only *texture memory*. It can accelerate memory accesses when neighboring threads read from nearby addresses.

2.2 Sparse-Matrix–Vector product on GPUs

Sparse-matrix computations pose some difficulties on GPUs, such as irregular memory accesses, load balancing and low cache efficiency. Several papers have focused on choosing suitable matrix formats and appropriate kernels to overcome the irregularity of the sparse matrix [4,26]. These works have explored implementing efficiently SpMV over real numbers. Schmidt et al. [19] proposed an optimized matrix format to accelerate exact SpMV over GF(2), that can be used in the linear algebra step of the Number Field Sieve (NFS) for integer factorization [22]. Boyer et al. [8] have adapted SpMV kernels over small finite fields and rings $\mathbb{Z}/m\mathbb{Z}$, where they used double-precision floating-point numbers to represent ring elements. In our context, since the order of the considered finite ring is large (hundreds of bits), specific computing models and addressing models should be used.

In this work, we have a prime ℓ, along with an N-by-N sparse matrix A defined over $\mathbb{Z}$, and we want to solve the linear system $Aw = 0$ over $\mathbb{Z}/\ell\mathbb{Z}$. A feature of the index calculus context that we consider here, is that A contains small values (e.g. 32-bit integers). In fact, around 90% of the non-zero coefficients are ± 1.

The very first columns of A are relatively dense, then the column density decreases gradually. The row density does not change significantly. We denote by n_{NZ} the

Fig. 1: Distribution of non-zero elements in an FFS matrix

number of non-zero elements in A. See Figure 1 for a typical density plot of a matrix arising in an FFS computation.

We will use the Wiedemann algorithm as a solver. This algorithm iterates a very large number of matrix-vector products of the form $v \leftarrow Au$, where u and v are dense N-coordinate vectors. The major part of this work deals with how to accelerate this product.

In order to carry out this product, we compute the dot product between each row of A and the vector u. The basic operation is of the form $x \leftarrow (x+\lambda y) \bmod \ell$, where λ is a non-zero coefficient of A, and x and y are coordinates of the vectors v and u, respectively. To minimize the number of costly reductions modulo ℓ, we accumulate computations, and postpone the final modular reduction of the result as late as possible. When iterating many products (computations of the form $A^i u$), we can further accumulate several SpMVs before reducing modulo ℓ, as long as the intermediate results do not exceed the largest representable integer. As far as arithmetic over $\mathbb{Z}/\ell\mathbb{Z}$ is concerned, we chose to use the Residue Number System, which appears to be more suited to the fine grained parallelism inherent to the SPMT computing model than the usual multi-precision representation of large integers. A comparison of the two representations is given in Subsection 6.3.

3 Residue Number System and Modular Arithmetic

3.1 A brief reminder on RNS

The Residue Number System (RNS) is based on the Chinese Remainder Theorem (CRT). Let $\mathcal{B} = (p_1, p_2, \ldots, p_n)$ be a set of mutually coprime integers, which we call an *RNS-basis*. We define P as the product of all the p_i's. The RNS uses the fact that any integer x within $[0, P-1]$ can be uniquely represented by the list $(x_1, x_2, \ldots, x_n)$, where each x_i is the residue of x modulo p_i, which we write as $x_i = |x|_{p_i}$.

If x and y are given in their RNS representations $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_n)$, according to $\mathcal{B}$, and such that $x, y < P$, RNS addition and multiplication are realized by modular addition and multiplication on each component:

$$\vec{x} \vec{+} \vec{y} = (|x_1 + y_1|_{p_1}, \ldots, |x_n + y_n|_{p_n}), \quad \vec{x} \vec{\times} \vec{y} = (|x_1 \times y_1|_{p_1}, \ldots, |x_n \times y_n|_{p_n})$$

The result (e.g., $x + y$) should belong to the interval $[0, P-1]$ if we want to obtain a valid RNS representation. Otherwise, it will be reduced modulo P. Unlike addition or multiplication, other operations such as comparison or modular reduction are more subtle in RNS.

We can convert back an RNS vector to the integer form by using the CRT formula:

$$x = \left| \sum_{i=1}^{n} x_i \cdot \left| P_i^{-1} \right|_{p_i} \cdot P_i \right|_P, \text{where } P_i \triangleq P/p_i.$$

This number system is particularly interesting for arithmetic over large integers, since it distributes the computation over several small residues. In other

words, the computation units that will work on the residues are independent and need no synchronization nor communication, as there is no carry propagation [23,24].

3.2 RNS reduction modulo ℓ

In the chosen RNS representation, $(P-1)$ is the largest representable integer. So in the case of repeated SpMVs over $\mathbb{Z}/\ell\mathbb{Z}$, we can accumulate at most $\log(\frac{P-1}{\ell-1})/\log(r)$ matrix–vector products before having to reduce modulo ℓ, where r corresponds to the largest row norm (defined as the sum of the absolute values of its elements) in the matrix. To reduce the vector v modulo ℓ, we use the method introduced by Bernstein in [6], which allows us to perform the reduction without having to convert the vector back to the integer form.

We assume that the RNS-basis $\mathcal{B}$ contains n moduli $p_1, \ldots, p_n$ of k bits each. We impose that the p_i's are close to 2^k. The reasons will be detailed in the following subsection. We want to reduce modulo ℓ an RNS vector $(x_1, \ldots, x_n)$. We start from the CRT reconstruction: $x = \left| \sum_{i=1}^{n} \gamma_i P_i \right|_P$, where we have defined $\gamma_i \triangleq \left| x_i {P_i}^{-1} \right|_{p_i}$. Let us also define the integer α as follows

$$\alpha = \left\lfloor \sum_{i=1}^{n} \frac{\gamma_i P_i}{P} \right\rfloor = \left\lfloor \sum_{i=1}^{n} \frac{\gamma_i}{p_i} \right\rfloor . \tag{1}$$

The vector x can then be written as $\sum_{i=1}^{n} \gamma_i P_i - \alpha P$ and, since $\gamma_i < p_i$, we have $0 \leq \alpha < n$.

Now, if we assume that α is known, we define $z \triangleq \sum_{i=1}^{n} \gamma_i \left|P_i\right|_\ell - \left|\alpha P\right|_\ell$. We can easily check that z is congruent to x modulo ℓ and lies in the interval $[0, \ell \sum_{i=1}^{n} p_i[$.

What remains to be done is to determine α. Since $p_i \approx 2^k$, we approximate the quotient γ_i/p_i using only the s most significant bits of $\gamma_i/2^k$. Hence, an estimate for α is proposed as

$$\hat{\alpha} \triangleq \left\lfloor \sum_{i=1}^{n} \frac{\left\lfloor \frac{\gamma_i}{2^{k-s}} \right\rfloor}{2^s} + \Delta \right\rfloor , \tag{2}$$

where s is an integer parameter in $[1, k]$ and Δ an error correcting term in $]0, 1[$.

Bernstein states in [6] that if $0 \leq x < (1-\Delta)P$ and $(\epsilon + \delta) \leq \Delta < 1$ where $\epsilon \triangleq \sum_{i=1}^{n} \frac{c_i}{2^k}$ and $\delta \triangleq n\frac{2^{k-s}-1}{2^k}$, then $\alpha = \hat{\alpha}$.

Once α is determined, we are able to perform an RNS computation of z. Algorithm 1 summarizes the steps of the computation.

Algorithm 1: Approximate RNS modular reduction

Precomputed: Vector $(\left|{P_j}^{-1}\right|_{p_j})$ for $j \in \{1, \ldots, n\}$
Table of RNS vectors of $|P_i|_\ell$ for $i \in \{1, \ldots, n\}$
Table of RNS vectors of $|\alpha P|_\ell$ for $\alpha \in \{1, \ldots, n-1\}$
Input : RNS vector of x, with $0 \leq x < (1 - \Delta)P$
Output : RNS vector of $z \equiv x \pmod{\ell}$, $z < \ell \sum_{i=1}^{n} p_i$

1 **foreach** *thread* j **do**
2 $\quad \gamma_j \leftarrow \left| x_j \times \left|{P_j}^{-1}\right|_{p_j} \right|_{p_j}$ `/* 1 RNS product */`
3 Broadcast of the γ_j's by all the threads
4 **foreach** *thread* j **do**
5 $\quad z_j \leftarrow \left| \sum_{i=1}^{n} \gamma_i \times \left||P_i|_\ell\right|_{p_j} \right|_{p_j}$ `/* (n − 1) RNS sums & n RNS products */`
6 $\quad \alpha \leftarrow \left\lfloor \sum_{i=1}^{n} \frac{\left\lfloor \frac{\gamma_i}{2^{k-s}} \right\rfloor}{2^s} + \Delta \right\rfloor$ `/* sum of n s-bit terms */`
7 $\quad z_j \leftarrow \left| z_j - \left||\alpha P|_\ell\right|_{p_j} \right|_{p_j}$ `/* 1 RNS subtraction */`

All the operations can be evaluated in parallel on the residues, except for step 3, where a broadcast of all the γ_j's is needed. Even if the obtained result z is not the exact reduction of x, it is bounded by $n2^k\ell$. Thus, we guarantee that the intermediate results of the SpMV computation do not exceed a certain bound less than P. Notice that this RNS reduction algorithm imposes that P be one modulus (k bits) larger than implied by the earlier condition $\ell < P$.

In conclusion, P is chosen, such that $r \times n2^k\ell < (1 - \Delta)P$, with r is the largest row norm of the matrix.

3.3 Modular reduction modulo p_j

The basic RNS operation is $z_j \leftarrow (x_j + \lambda \times y_j) \bmod p_j$, where $0 \leq x_j, y_j, z_j < p_j$ are RNS residues and λ is a positive element of the matrix. So, it consists of an AddMul (multiplication, then an addition) followed by a reduction modulo p_j. To speed up the reduction modulo p_j, the moduli are chosen of the pseudo-Mersenne form $2^k - c_j$, with c_j as small as possible.

In fact, let us define $t_j \triangleq x_j + \lambda \times y_j$ as the intermediate result before the modular reduction. t_j can be written as

$$t_j = t_{jL} + 2^k \times t_{jH}, \text{where } t_{jL} \triangleq t_j \bmod 2^k, t_{jH} \triangleq t_j/2^k. \tag{3}$$

Since $2^k \equiv c_j \pmod{p_j}$, we have $t_j \equiv t_{jL} + t_{jH} \times c_j \pmod{p_j}$. So, we compute $t_j \leftarrow t_{jL} + t_{jH} \times c_j$, then we have to consider two cases:

- if $t_j < 2^k$, we have "almost" reduced $(x_j + \lambda \times y_j)$ modulo p_j, since the result lies in $[0, 2^k[$, not in $[0, p_j[$;

– else we have reduced t_j by approximately k bits. Thus, we repeat the previous procedure with $t_j \leftarrow t_{jL} + c_j \times t_{jH}$, which then satisfies $t_j < 2^k$.

The output lies in $[0, 2^k - 1]$, so we propose to relax the condition on both input and output: $x_j, z_j \in [0, 2^k - 1]$. With this approach, the reduction can be done in a small number of additions and products.

3.4 Possible RNS Mappings on GPU/CPU

We represent the finite ring $\mathbb{Z}/\ell\mathbb{Z}$ as the integer interval $[0, \ell - 1]$. Each element is stored in its RNS form. On GPU, we opted for 64-bit moduli (i.e. $k = 64$), for performance considerations. Even that floating point instructions have higher throughput, integer instructions gave better performances, because with floating point arithmetic, only the mantissa is used and the algorithms are more complex than with integer arithmetic. We use the PTX (*parallel thread execution*) pseudo-assembly language for CUDA [15] to implement the RNS operations.

On CPU, we implemented three versions based on:
– MMX instruction set: we map an RNS residue to an unsigned 64-bit integer.
– Streaming SIMD Extensions (SSE2) set: a 128-bit XMM register holds two residues, so the processor can process two residues simultaneously.
– Advanced Vector Extensions (AVX2) set: we use the 256-bit YMM register to hold four residues.

4 Sparse Matrix Storage Formats

In this section, we assume that the elements of the matrix, as well as the elements of the vectors u and v are in a field K (reals, finite fields, etc.). For each format, we will discuss how to perform the matrix-vector product. We will give a pseudo-code for the format CSR. Figures that illustrate the other formats and their corresponding Pseudo-code can be found in Appendix A.

The matrix and vectors are put in *global memory*, since their sizes are important. Temporary results are stored in registers. The *shared memory* is used when partial results of different threads are combined. Arithmetic operations are performed in registers and denoted in the pseudo-code by the function `addmul()`.

Coordinate (COO)

The format COO consists of three arrays `row_id`, `col_id` and `data` of n_{NZ} elements. The row index, column index and the value are explicitly stored to specify a non-zero matrix coefficient. In this work, we propose to sort the matrix coefficients by their row index.

A typical way to work with the COO format on GPU is to assign one thread to each non-zero matrix coefficient. This implies that different threads from different warps will process a same row. Each thread computes its partial result, then performs a segmented reduction [7,20] to sum the partial results of the other threads belonging to the same warp and spanning the same row. We followed the scheme proposed by the library CUSP [5], which performs the segmented reduction in shared memory, using the row indices as segment descriptors. Each

warp iterates over its interval, processing 32 coefficients at a time. If a spanned row is fully processed, its result is written to v, otherwise, the row index and the partial dot product are stored in temporary arrays. Then, a second kernel performs the combination of the per-warp results.

The main drawbacks of the COO kernel are the cost of the combination of partial results and excessive usage of *global memory*. Its advantage is that the workload distribution is balanced across warps, as they iterate over a constant length interval.

Compressed Sparse Row (CSR)
The CSR format stores the column indices and the values of the non-zero elements of A into two arrays of n_{NZ} elements: `id` and `data`. A third array of pointers, `ptr`, of length $N+1$, is used to indicate the beginning and the end of each row. Non-zero coefficients are sorted by their row index. The CSR format eliminates the explicit storage of the row index, and is convenient for a direct access to the matrix, since `ptr` indicates where each row starts and ends in the other two ordered arrays.

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) Sparse matrix A

$$\texttt{data} = \begin{bmatrix} a_{01} & a_{03} & a_{11} & a_{14} & a_{15} & a_{20} & a_{22} & a_{23} & \dots \end{bmatrix}$$

$$\texttt{id} = \begin{bmatrix} 1 & 3 & 1 & 4 & 5 & 0 & 2 & 3 & \dots \end{bmatrix}$$

$$\texttt{ptr} = \begin{bmatrix} 0 & 2 & 5 & 8 & \dots \end{bmatrix}$$

(b) CSR representation

Scalar approach (CSR-S) To parallelize the product for the CSR format, a simple way is to assign each row to a single thread (*scalar* approach). For each non-zero coefficient, the thread performs a read from *global memory*, an `addmul` and a write in *registers*. Final result is written to *global memory*.

Vector approach (CSR-V) The *vector* approach consists in assigning a warp to each row of the matrix [4]. The threads within a warp access neighboring non-zero elements, which makes the warp accesses to `id` and `data` contiguous. Each thread computes its partial result in shared memory, then a parallel reduction in shared memory is required to combine the per-thread results (denoted `reduction_csr_v()` in Algorithm 2). No synchronization is needed, since threads belonging to a same warp are implicitly synchronized.

Compared to COO kernel, the two CSR kernels reduce the usage of *global memory* and simplify the combination of partial results. The CSR kernels suffer from load unbalance, if the rows have widely varying lengths. To improve the load balance, one possibility is to order the rows by their lengths. So, the warps launched simultaneously have almost the same load.

If we compare the two CSR kernels. The threads of CSR-S have non contiguous access to `data` et `id`, as they do not work on the same rows. Thus, their memory accesses are not as efficient as the accesses of the CSR-V. However, the

Algorithm 2: CSR-V for row i executed by thread of index `tid` in its warp

Inputs : `data`: array of n_{NZ} elements of K, `id`: array of n_{NZ} positive integers, `ptr`: array of N positive integers and u: vector of N elements of K.
Output: v: vector of N elements of K.

```
sum ← 0;
j ← ptr_i + tid;                    // position of beginning for each thread
While j < ptr_{i+1} do
    sum ← addmul(sum,data_j ,u_{id_j});
    j ← j + 32;
reduction_csr_v(sum,tid);                    // reduction in shared memory
If tid= 0 then    // first thread of the warp writes in global memory
    v_i ← sum;
```

CSR-V kernel requires a combination of partial results which increases the use of *registers* and *shared memory* (cf. Subsection 6.2).

ELLPACK (ELL)
The ELL format extends the CSR arrays to N-by-K arrays, where K corresponds to the maximum number of non-zero coefficients per row. The rows that have less than K non-zero coefficients are padded. Since the padded rows have the same length, only column indices are explicitly stored. This format suffers from the overhead due to the padding when the the percentage of zeros is high. An optimization was proposed by Vázquez et al. with a format called ELLPACK-R (ELL-R) [26]. This variant adds an array `len` of length N that indicates the length of each row. Thus, the zeros added by the padding are not considered when performing the matrix-vector product.

The partitioning of the work is done by assigning a thread to a row of the matrix. The kernel takes advantage from the column-major ordering of the elements to improve the accesses on the vector u. However, it suffers from thread divergence.

Sliced Coordinate (SLCOO)
The SLCOO format was introduced on GPUs by Schmidt et al. for integer factorization, in the particular case of matrices over GF(2) [19] and was inspired by the CADO-NFS [2] software for CPUs. The aim of this format is to increase the cache hit rate that limits the CSR and COO performance. Like COO, the SLCOO representation stores the row indices, column indices and values. However, it divides the matrix into horizontal slices, where the non-zero coefficients of a slice are sorted according to their column index in order to reduce the irregular accesses on source vector u, if they had been sorted by their row indices. A fourth array `ptrSlice` indicates the beginning and end of each slice. We denote this format SLCOO-σ, where the parameter σ is the number of rows in a slice.

For the SLCOO kernel, each warp works on a slice. Since each thread works on more than one row, it needs to have individual storage for its partial per-row results, or to be able to have exclusive access to a common resource. In [19],

Schmidt et al. mentionned the two possibilities of either using the *shared memory* or having atomic accesses. While these needs can be fulfilled in [19] for the context of linear algebra over GF(2), we will observe in Section 6 that these constraints hamper the efficiency of the SLCOO in the context of large fields.

There are other SpMV formats in the literature, such as DIA (Diagonal) format, that are appropriate only for matrices that satisfy some sparsity patterns, which is not our case.

5 SpMV Kernels over large fields

In the context of our application, the matrix elements are "small" (32 bit integers) and the vectors elements are in $\mathbb{Z}/\ell\mathbb{Z}$. In this section, we study how we adapt the kernels to this context. We assume that an element of $\mathbb{Z}/\ell\mathbb{Z}$ holds in n machine words. Thus, processing a non-zero coefficient λ at row i and column j of the matrix implies reading the n words that compose the j^{th} element in the input vector u, multiply them by λ and adding them to the n words that compose the i^{th} element in the output vector v. In the following pseudo-code, we denote the arithmetic operation that applies to a word by the function `addmul_word()`. Pseudo-code is therefore given without details regarding the representation system of the numbers and the resulting arithmetic.

***Sequential* Scheme** A first approach would be that each thread processes a coefficient. We would call this scheme *sequential*. To illustrate this scheme, we apply it on the CSR-V kernel.

Algorithm 3: CSR-V-seq for row i executed by thread of index `tid` in its warp

Inputs : `data`: array of n_{NZ} signed integers, `id`: array of n_{NZ} positive integers, `ptr`: array of N positive integers, u: vector of $N \times n$ machine words.
Output: v: vector of $N \times n$ machine words.

```
Declare array sum ← {0};                    // n machine words initialized to 0
j ← ptr_i + tid;
While j < ptr_{i+1} do
    For k ← 0 to n do
        sum_k ← addmul_word(sum_k, data_j, u_{id_j × n + k});    // process k^th word
    j ← j + 32;
reduction_csr_v_seq(sum,tid);
If tid= 0 then
    For k ← 0 to n do
        v_{i×n+k} ← sum_k;                  // store k^th word in global memory
```

This scheme suffers from several drawbacks. The first one is that the thread processes the n machine words corresponding to the coefficient, i.e. reads and writes the n machine words and makes n arithmetic operations. Thus, the thread consumes more *registers*. The second is that the threads of the same warp access non-contiguous zones of the vectors u and v, as their accesses are always spaced by n words.

***Parallel* Scheme** To overcome the limitations of the previous approach, a better scheme would be that a nonzero coefficient is processed by n threads. We refer to the scheme by the *parallel* scheme. Threads of the same warp are organized in n_{GPS} of n threads, where $n_{\text{GPS}} \times n$ is closest to 32, the number of threads per warp. Each group is associated to a non-zero matrix coefficient. For example, for n = 5, we take $n_{\text{GPS}} = 6$, so the first 5 threads process in parallel the 5 words of the 1st source vector element, threads 5 to 9, process the words of the 2nd source vector element, and so on, and we will have two idle threads per warp.

Algorithm 4: CSR-V-par for row i executed by thread of index `tid` in its warp

Inputs : `data`: array of n_{NZ} signed integers, `id`: array of n_{NZ} positive integers, `ptr`: array of N positive integers, u: vector of $N \times n$ mots machines.
Output: v: vector of $N \times n$ mots machines.

```
sum ← 0;                                        // 1 machine word initialized to 0
j ← ptr_i + ⌊ tid / n ⌋;                  // position of beginning for each thread
While j < ptr_{i+1} do
    sum ← addmul_word(sum,data_j ,u_{id_j × n + tid mod n});    // process 1 word
    j ← j + n_GPS;
reduction_csr_v_par(sum,tid);
If tid< n then     // first group of the warp writes in global memory
    v_{i × n + tid} ← sum;
```

For the other kernels, both schemes are applicable and the *parallel* scheme always performs significantly better than the *sequential* scheme.

6 Comparative Analysis of SpMV Kernels

In this section, we compare the performances of the kernels that we presented. The objective is to minimize the time of a matrix-vector product. The experiments were run on an NVIDIA GeForce GTX 680 graphics processor (Kepler). Each SpMV kernel was executed 100 times, which is large enough to obtain stable timings. Our measurements do not include the time spent to copy data between the host and the GPU, since the matrix and vectors do not need to be transferred back and forth between each SpMV iteration. The reduction modulo ℓ happens only once every few iterations, which is why the timing of an iteration includes the timing of the reduction modulo ℓ kernel multiplied by the frequency of its invocation. The reported measurements are based on NVIDIA developer tools.

Table 1 summarizes the considered matrix over $\mathbb{Z}/\ell\mathbb{Z}$. The matrix was obtained during the resolution of discrete logarithm problem in the 217-bit prime order subgroup of $\text{GF}(2^{619})^{\times}$ using the FFS algorithm. The $\mathbb{Z}/\ell\mathbb{Z}$ elements fit in four RNS 64-bit residues. Since, an extra

Size of the matrix (N)	650k × 650k
#Non-zero coefficients	65M
Max (row norm)	492
Percentage of ± 1	92.7%
Size of ℓ (in bits)	217
Size of M (in bits)	320
Size of $n2^k\ell$ (in bits)	283
Frequency of reduction modℓ	1/4

Table 1: Properties of test matrix

residue is needed for the modular reduction (cf. Subsection 3.2), the total number of RNS residues is $n = 5$.

6.1 Comparison of schemes *sequential* and *parallel*

We compare the application of the two schemes on the CSR-V kernel. The *sequential* kernel consumes more *registers* and *shared memory*, which limits the maximum number of warps that can be run on a SM to 24. In our application, CSR-S was limited to 24 warps/SM, for a bound of 64 warps/SM. This is reported in the column *Theoretical Occupancy* of the following table. The low occupancy significantly decreases the performance. Concerning the *global memory* access pattern, the column *Load/Store efficiency* gives the ratio of requested memory transactions to the number of transactions performed, which reflects the degree to which memory accesses are coalesced (100% efficiency) or not. For *sequential* kernel, uncoalesced accesses cause the bandwidth loss and the performance degradation. The *parallel* kernel makes the write accesses coalesced (100% store efficiency). For the loads, it reaches only 47% due to irregular accesses on the source vector.

	Registers per thread	Shared Memory per SM	(Theoretical) Occupancy	Load / Store efficiency	Timing in ms
Sequential	27	49152	35.1% (37.5%)	7.5% / 26%	141.1
Parallel	**21**	**15360**	**70.3% (100%)**	**47.2% / 100%**	**41.4**

It is clear that the *parallel* scheme is better suited to the context of large integers. We apply this scheme to other formats and compare their performance in the following subsection.

6.2 Comparison of kernels CSR, COO, ELL and SLCOO

Due to the segmented reduction, the COO kernel performs more instructions and requires more registers. Thread divergence happens more often, because of the several branches that threads belonging to the same warp can take.

As far as the ELL kernel is concerned, the padded rows have the same length. This yields a good balancing across the warps and the threads (cf. *Occupancy* and *Branch divergence* in the following table). The column-major ordering makes this kernel reach the highest cache hit rate.

The CSR-S kernel suffers from low efficiency of memory access compared to CSR-V. In fact, with the kernel CSR-S, the threads with the same warp work on several lines simultaneously, which makes their access to tables `id` and `data` not contiguous. The kernel CSR-V better satisfies the GPU architectural specificities.

	Registers per thread	Branch divergence	(Theoretical) Occupancy	Load / Store efficiency	Cache hit rate	Timing in ms
COO	25	47.1%	(66.7%) 65.2%	34.3%/37.8%	34.4%	88.9
CSR-S	18	28.1%	(100%) 71.8%	29.2%/42.5%	35.4%	72.3
CSR-V	**21**	**36.7%**	**(100%) 70.3%**	**47.2%/100%**	**35.4%**	**41.4**
ELL-R	18	44.1%	(100%) 71.8%	38.1%/42.5%	40.1%	45.5

The next table compares the CSR-V kernel with several SLCOO kernels for different slice sizes. We remark that increasing the slice size improves the cache hit rate, since accesses on the source vector are less irregular. However, by making the slices larger, we increase the usage of shared memory proportionally to the slice size, which limits the maximum number of blocks that can run concurrently. This limitation of the occupancy yields poor performance compared to the CSR-V kernel. Here, we consider an L1-oriented configuration (48kB L1, 16kB shared) of the on-chip memory. It is possible to move to a shared-oriented configuration (16kB L1, 48kB shared). This improves the occupancy, but degrades the cache hit rate, and finally does not improve the performances.

	Shared Memory per Block	Blocks per SM	(Theoretical) Occupancy	Cache hit rate	Timing in ms
CSR-V	**1920**	**8**	**(100%) 70.3%**	**35.4%**	**41.4**
SLCOO-2	3840	6	(75%) 68.4%	36.1%	46.9
SLCOO-4	7680	4	(50%) 44.5%	36.9%	58.6
SLCOO-8	15360	2	(22.5%) 22.3%	37.8%	89.9

Combinations of different formats have been tested. However they do not give better results. Splitting the matrix into column major blocks, or processing separately the first dense columns did not improve the performance either.

For our matrix, the main bottleneck is memory access. In the CSR-V kernel, 72% of the time is spent in reading data, 2% in writing data and 26% in computations.

6.3 Comparison of RNS and Multi-precision arithmetics

We also implemented the RNS and the multi-precision (MP) arithmetics on GPU. For the MP representation, to perform the reduction modulo ℓ, we use a precomputed inverse of ℓ so as to divide by ℓ using a single multiplication. For the 217-bit prime order subgroup, choosing the largest representable integer $M = 2^{256} - 1$ is sufficient to accumulate a few number of SpMVs before reducing modulo ℓ. In fact, the maximum row norm that we have (492) allows to do up to 4 iterative SpMVs before having to reduce.

For MP kernel, the reduction kernel takes only 0.37 ms, which corresponds to less than 0.1 ms per iteration. In RNS, we can accumulate 4 SpMVs before the reduction modulo ℓ (cf. Table 1). The reduction kernel takes 1.6 ms (i.e., 0.4 ms per iteration).

The idea behind the use of RNS rather than MP arithmetic is that RNS can significantly decrease data sharing between the threads and arithmetic operations required for the carry generation/propagation. The RNS kernel allows us to reach higher occupancy and better performance. The speed-up of RNS compared to multi-precision on the SpMV timing is around 15%.

	Registers per thread	Shared Memory per Block	Executed Instructions	(Theoretical) Occupancy	Timing in ms
MP	21	2880	6.1×10^8	(83.3%) 51.2%	46.6
RNS	**18**	**1920**	5.8×10^8	**(100%) 70.3%**	**41.4**

7 Improvements on CSR-V Kernel

To further improve the kernel performance, one should take into account the GPU architectural characteristics: the management of the memory accesses, the partitioning of the computations and the specificities of the problem considered.

Texture caching Although our SpMV kernel suffers from irregular load accesses, a thread is likely to read from an address near the addresses that nearby threads (of the same group) read. For this reason, we bind on texture memory and replace reads with texture fetches. This improves the global memory efficiency and consequently the SpMV delay.

Reordering the non-zero coefficients of a row Since most of the coefficients of the matrix are ± 1, it seems promising to treat multiplications by these coefficients differently from other coefficients: additions and subtractions are less expensive than multiplications. All these separations result in code divergence, that we fix by reordering the non-zero coefficients in the matrix such that values of the same category $(+1, -1, > 0, < 0)$ are contiguous. This decreases the branch divergence and decreases the total SpMV delay.

Compressing the values array `data` Since the majority of the coefficients are ± 1, after reordering the coefficients per row, we can replace the ± 1 coefficients by their occurrence count. This reduces the length of the values array `data` by more than 10 times, and so reduces the number of reads.

Improving warp balancing In the CSR-V kernel, each warp processes a single row. This requires launching a large number of warps. Consequently, there is a delay to schedule those launched warps. Instead, we propose that each warp iterates over a certain number of rows. To further increase the occupancy, we permute the rows such that each warp roughly gets the same work load.

	Performance Effects	Timing in ms (speedup)
Texture caching	Global Load efficiency: 47.2% → 84%	32 (+30%)
Non-zeros reordering	Branch Divergence: 36.7% → 12.9%	30.5 (+5%)
Compressing `data`	Executed Instructions ($\times 10^8$): 5.8 → 5.72	27.6 (+11%)
Multiple iterations	Occupancy: 70.3% → 74.9%	27.4 (+0.5%)
Rows permutation	Occupancy: 74.9% → 81.8%	**27.1** (+1%)

8 Reference Software Implementation

For comparison purposes, we implemented SpMV on the three software instruction set architectures MMX, SSE and AVX, based on the RNS representation for the arithmetic and the CSR format for the storage of the matrix. We have not explored other formats that can be suitable for CPU. Probably blocked formats that better use the cache can further improve the performance on CPU.

Unlike for GPU, processing separately the first dense columns accelerates the CPU SpMV of around 5%.

We can report the computational throughput in terms of GFLOP/s, which we determine by dividing the number of required operations (twice the number of non-zero elements in the matrix A multiplied by $2 \times n$) by the running time. We will use this unit of measure for the throughput even for integer instructions.

The experiment was run on an Intel CPU i5-4570 (3.2 GHz) using 4 threads on 4 cores. The AVX2 implementation using integers is the fastest implementation and reaches the highest throughput. However, the fact that the number of moduli is a multiple of four entail overheads. When comparing the software performance with the GPU one, the fastest software implementation is 4 to 5 times slower than on one graphics processor.

	Length of modulus	Number of moduli (n)	Timing in ms	Throughput in GFLOP/s
MMX (integer)	64	5	306	5.3
MMX (double-precision floats)	52	6	351	3.7
SSE2 (integer)	63	6	154	12.1
SSE2 (double-precision floats)	52	6	176	10.6
AVX2 (integer)	**63**	**8**	**117**	**21.2**
AVX2 (double-precision floats)	52	8	135	18.3
GPU (integer)	**64**	**5**	**27**	**57.6**

9 Conclusion

We have investigated different data structures to perform iterative SpMV for DLP matrices on GPUs. We have adapted the kernels for the context of large finite fields and added optimizations suitable to the sparsity and the specific computing model. The CSR-V kernel based on the *parallel* scheme appears to be the most efficient one. The SLCOO poses for the sizes that we use some hardware difficulties that nullify its contribution on increasing the cache hit rate. Future GPUs may enhance the performance. We have shown that using RNS for finite field arithmetic provides a considerable degree of independence, which can be exploited by massively parallel hardware. This implementation contributed to solving the discrete logarithm problem in GF(2^{619}) and GF(2^{809}) (See Appendix B and [3,10] for further details).

References

1. L. Adleman. A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 55–60, Washington, DC, USA, 1979.
2. S. Bai, C. Bouvier, A. Filbois, P. Gaudry, L. Imbert, A. Kruppa, F. Morain, E. Thomé, and P. Zimmermann. Cado-nfs: Crible algébrique: Distribution, optimisation - number field sieve. http://cado-nfs.gforge.inria.fr/.
3. 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 *PKC 2014*, pages 221–238. Springer Berlin Heidelberg, 2014.

4. N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.
5. N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. http://code.google.com/p/cusp-library/.
6. D. J. Bernstein. Multidigit modular multiplication with the explicit chinese remainder theorem. Technical report, 1995. http://cr.yp.to/papers/mmecrt.pdf.
7. G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, August 1993.
8. B. Boyer, J.-G. Dumas, and P. Giorgi. Exact sparse matrix-vector multiplication on GPU's and multicore architectures. *CoRR*, abs/1004.3719, 2010.
9. T. Hayashi, T. Shimoyama, N. Shinohara, and T. Takagi. Breaking pairing-based cryptosystems using η_t pairing over GF(3^{97}). Cryptology ePrint Archive, Report 2012/345, 2012.
10. H. Jeljeli. Resolution of linear algebra for the discrete logarithm problem using gpu and multi-core architectures. In *Euro-Par 2014*, pages 764–775, 2014.
11. E. Kaltofen. Analysis of coppersmith's block wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):pp. 777–806, 1995.
12. B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. In *CRYPTO '90*, pages 109–133, London, UK, 1991. Springer-Verlag.
13. C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.
14. NVIDIA Corporation. *CUDA Programming Guide Version 4.2*, 2012. http://developer.nvidia.com/cuda-downloads.
15. NVIDIA Corporation. *PTX: Parallel Thread Execution ISA Version 3.0*, 2012. http://developer.nvidia.com/cuda-downloads.
16. A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology*, 1984.
17. J. M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15:331–334, 1975.
18. C. Pomerance and J. W. Smith. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experiment. Math*, 1:89–94, 1992.
19. B. Schmidt, H. Aribowo, and H.-V. Dang. Iterative sparse matrix-vector multiplication for integer factorization on GPUs. In *Euro-Par 2011 Parallel Processing*, volume 6853, pages 413–424. Springer, 2011.
20. S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. pages 97–106, August 2007.
21. D. Shanks. Class number, a theory of factorization, and genera. In *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*, pages 415–440. Providence, R.I., 1971.
22. P. Stach. Optimizations to nfs linear algebra. *CADO workshop on integer factorization*. http://cado.gforge.inria.fr/workshop/abstracts.html.
23. N. S. Szabo and R. I. Tanaka. *Residue Arithmetic and Its Applications to Computer Technology*. McGraw-Hill Book Company, New York, 1967.
24. F. J. Taylor. Residue arithmetic a tutorial with examples. *Computer*, 17:50–62, May 1984.
25. E. Thomé. Subquadratic computation of vector generating polynomials and improvement of the block wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757 – 775, 2002.

26. F. Vázquez, E. M. Garzón, J. A. Martinez, and J. J. Fernández. The sparse matrix vector product on GPUs. Technical report, University of Almeria, June 2009.
27. D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theor.*, 32(1):54–62, January 1986.

A Formats and GPU Kernels of SpMV

$$\begin{pmatrix} 0 & a_{01} & 0 & a_{03} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{14} & a_{15} \\ a_{20} & 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{34} & 0 \\ 0 & a_{41} & a_{42} & 0 & 0 & a_{45} \\ 0 & 0 & a_{52} & 0 & 0 & a_{55} \end{pmatrix}$$

(a) Sparse matrix A

$$\texttt{data} = \begin{bmatrix} a_{01} & a_{03} & * \\ a_{11} & a_{14} & a_{15} \\ a_{20} & a_{22} & a_{23} \\ a_{31} & a_{34} & * \\ a_{41} & a_{42} & a_{45} \\ a_{52} & a_{55} & * \end{bmatrix} \quad \texttt{id} = \begin{bmatrix} 1 & 3 & * \\ 1 & 4 & 5 \\ 0 & 2 & 3 \\ 1 & 4 & * \\ 1 & 2 & 5 \\ 2 & 5 & * \end{bmatrix} \quad \texttt{len} = \begin{bmatrix} 2 & 3 & 3 & 2 & 3 & 2 \end{bmatrix}$$

(b) ELL-R representation

$$\begin{aligned} \texttt{data} &= \begin{bmatrix} a_{01} & a_{11} & a_{03} & a_{14} & a_{15} & a_{20} & a_{31} & a_{22} & a_{23} & a_{34} & \dots \end{bmatrix} \\ \texttt{row\_id} &= \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 2 & 3 & 2 & 2 & 3 & \dots \end{bmatrix} \\ \texttt{col\_id} &= \begin{bmatrix} 1 & 1 & 3 & 4 & 5 & 0 & 1 & 2 & 3 & 4 & \dots \end{bmatrix} \\ \texttt{ptrSlice} &= \begin{bmatrix} 0 & 5 & 10 & \dots \end{bmatrix} \end{aligned}$$

(c) SLCOO-2 representation

Algorithm 4: ELL-R for row i executed by one thread

Inputs : `data`: array of $K \times N$ elements of K, `id`: array of $K \times N$ positive integers,
`len`: array of N positive integers and u: vector of N elements of K.
Output: v: vector of N elements of K.

`sum` $\leftarrow 0$;
For $j \leftarrow 0$ **to** $\texttt{len}_i$ **do**
  `sum` $\leftarrow$ `addmul(sum,data`$_{N \times j + \texttt{row}}$`,`$u_{\texttt{id}_{N \times j + \texttt{row}}}$`)`;
$v_i \leftarrow$ `sum`;

Algorithm 5: SLCOO-σ for slice i executed by thread of index `tid` in its warp

Inputs : `data`: array of n_{NZ} elements of K, `row_id`, `col_id`: arrays of n_{NZ} positive integers, `ptrSlice`: array of N positive integers and u: vector of N elements of K.
Output: v: vector of N elements of K.

Declare array `sum` $\leftarrow \{0\}$; `// array of` σ `elements of` K
$i \leftarrow \texttt{ptrSlice}_i + \texttt{tid}$; `// position of beginning for each thread`
While $j < \texttt{ptrSlice}_{i+1}$ **do**
  $\texttt{sum}_{\texttt{row\_id}_j \bmod \sigma} \leftarrow \texttt{addmul}(\texttt{sum}_{\texttt{row\_id}_j \bmod \sigma}, \texttt{data}_j, u_{\texttt{col\_id}_j \bmod \sigma})$;
  $j \leftarrow j + 32$;
`reduction_slcoo(sum,tid);` `// reduction in` ***shared memory***
If `tid`$= 0$ **then** `// first thread of warp writes in` ***global memory***
  For $j \leftarrow 0$ **to** σ **do**
    $v_{i \times \sigma + j} \leftarrow \texttt{sum}_j$;

B Resolution of Linear Algebra of the Function Field Sieve

The linear algebra step consists of solving the system $Aw = 0$, where A is the matrix produced by the filtering step of the FFS algorithm. A is singular and square. Finding a vector of the kernel of the matrix is generally sufficient for the FFS algorithm.

The simple Wiedemann algorithm [27] which resolves such a system, is composed of three steps:

- *Scalar products*: It consists on the computation of a sequence of scalars $a_i = {}^t x A^i y$, where $0 \leq i \leq 2N$, and x and y are random vectors in $(\mathbb{Z}/\ell\mathbb{Z})^N$. We take x in the canonical basis, so that instead of performing a full dot product between ${}^t x$ and $A^i y$, we just store the element of $A^i y$ that corresponds to the non-zero coordinate of x.
- *Linear generator*: Using the Berlekamp-Massey algorithm, this step computes a linear generator of the a_i's. The output F is a polynomial whose coefficients lie in $\mathbb{Z}/\ell\mathbb{Z}$, and whose degree is very close to N.
- *Evaluation*: The last step computes $\sum_{i=0}^{deg(F)} A^i F_i y$, where F_i is the i^{th} coefficient of F. The result is with high probability a non-zero vector of the kernel of A.

The Block Wiedemann algorithm [11] proposes to use m random vectors for x and n random vectors for y. The sequence of scalars is thus replaced by a sequence of $m \times n$ matrices and the numbers of iterations of the first and third steps become $(N/n + N/m)$ and N/n, respectively. The n subsequences can be computed independently and in parallel. So, the block Wiedemann method allows to distribute the computation without an additional overhead [25].

B.1 Linear Algebra of FFS for GF(2^{619})

The matrix has 650k rows and columns. The prime ℓ is 217 bits. The computation was completed using the simple Wiedemann algorithm on a single NVIDIA GeForce GTX 580. The overall computation needed 16 GPU hours and 1 CPU hour.

B.2 Linear Algebra of FFS for GF(2^{809})

The matrix has 3.6M rows and columns. The prime ℓ is 202 bits. We run a Block Wiedemann on a cluster of 8 GPUs. We used 4 distinct nodes, each equipped with two NVIDIA Tesla M2050 graphics processors, and ran the Block Wiedemann algorithm with blocking parameters $m = 8$ and $n = 4$. The overall computation required 4.4 days in parallel on the 4 nodes.

These two computations were part of record-sized discrete logarithm computations in a prime-degree extension field [3,10].