Resolution of Linear Algebra for the Discrete Logarithm Problem using GPU and Multi-core Architectures

Hamza Jeljeli

Resolution of Linear Algebra for the Discrete Logarithm Problem using GPU and Multi-core Architectures

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 cryptanalysis, solving the discrete logarithm problem (DLP) is key to assessing the security of many public-key cryptosystems. The index-calculus methods, that attack the DLP in multiplicative subgroups of finite fields, require solving large sparse systems of linear equations modulo large primes. This article deals with how we can run this computation on GPU- and multi-core-based clusters, featuring InfiniBand networking. More specifically, we present the sparse linear algebra algorithms that are proposed in the literature, in particular the block Wiedemann algorithm. We discuss the parallelization of the central matrix–vector product operation from both algorithmic and practical points of view, and illustrate how our approach has contributed to the recent record-sized DLP computation in GF(2^{809}).

Keywords: Discrete logarithm problem, sparse linear algebra, parallel computing, GPU acceleration, multi-core processors, InfiniBand.

1 Introduction

The security of several public-key cryptosystems and protocols relies on the hardness of the computation of the discrete logarithm problem (DLP) in a given cyclic group [20]. To name but a few, we can mention the Diffie–Hellman key exchange protocol [11], the ElGamal encryption system [13] or the pairing-based cryptography [12].

In this context, a family of algorithms, known as *index-calculus* methods, is used to attack the DLP on finite fields. The majority of these algorithms propose to solve it in time sub-exponential in the size of the finite field. While a stream of recent algorithmic improvements for fields of small characteristic, including a quasi-polynomial algorithm [7], have produced several record-sized computations [22], the sub-exponential methods appear to be most competitive for fields of prime extension degree, at least so far.

Index calculus algorithms require solving large sparse systems of linear equations over finite fields. It is important to mention that, in this context, we are interested in exact linear algebra which is a completely distinct issue from numerical linear algebra. Hence, most considerations and methods in the case of numerical computations do not apply here. Several papers have focused on efficient implementations of sparse linear algebra over finite fields. For instance, Schmidt et al. [23] treated linear algebra over GF(2) for integer factorization; Boyer et al. [9] worked on the case of small finite rings and fields.

Problem Statement. Let GF(q) be the field in which the DLP is to be solved. The linear algebra is performed modulo a large prime ℓ that divides $q-1$. We consider ℓ between 160 and 650 bits, along with an N-by-N sparse matrix A defined over $\mathbb{Z}/\ell\mathbb{Z}$. The size N ranges from hundreds of thousands to millions. Each row of A contains $O\left((\log N)^2\right)$ non-zero coefficients. The very first columns of A are relatively dense, then the column density decreases gradually. The row density does not change significantly (cf. Figure 2). The so-called linear algebra step in the DLP computation consists in finding a non-trivial vector $w \in (\mathbb{Z}/\ell\mathbb{Z})^N$ such that $Aw = 0$.

We assume that we have access to one or several high-performance computing clusters, containing multi-core CPUs and/or GPUs, interconnected by fast communication links (typically InfiniBand). We want to optimize the use of these resources in order to solve the linear algebra problem efficiently. In particular, we aim to minimize the overall wall-clock time for solving the problem. First, at an algorithmic level, we study how these heavy computations can be distributed into smaller parallel subtasks. Then, we focus on more practical concerns, for instance the communication within these different subtasks.

Organization. This article is organized as follows: Section 2 gives an overview of the relevant algorithms for sparse linear algebra, while we discuss the parallelization of the matrix–vector product operation and focus on the communication concerns in Section 3. Finally, Section 4 details how our implementation has been used in concrete DLP computations with different hardware setups.

2 Algorithms for Sparse Linear Algebra

To solve systems of linear equations, two families of algorithms are available: *direct methods*, such as Gaussian elimination or LU/QR decompositions, and *iterative methods*, such as the conjugate gradient method and, in the context of linear algebra over finite fields, the Lanczos [19] and Wiedemann [26] algorithms.

The first set of algorithms requires $O\left(N^{\omega}\right)$ field operations, where ω is, for implementation concerns, 2.81 at best using the Strassen algorithm for matrix multiplication [24]. However, these methods tend to densify the matrix, which quickly raises storage issues. The second set of algorithms does not modify the matrix and requires $O\left(N\right)$ sparse-matrix–vector products (SpMVs). As long as an SpMV can be performed faster than $O\left(N^{\omega-1}\right)$ field operations, the iterative methods are asymptotically faster. This condition is reasonable, since the complexity of an SpMV is $O\left(N\gamma\right)$, where γ is the average number of non-zero coefficients per row. From both storage and complexity points of view, the iterative methods appear to be more suited to sparse linear algebra.

Still, in our case, despite the fact that the matrix is extremely sparse, the cost of an iterative solver remains high because the matrix is very large. The exact nature of the computation calls for no less than N iterations, or a number proportional to N depending on some fine points. The approach following is applied to tackle that problem. First, a structured Gaussian elimination (SGE) is run as a preprocessing step so as to reduce the size of the matrix [21]; then an iterative solver is used. Although the Gaussian elimination increases the average row weight, it nevertheless allows us to decrease the cost of the iterative solver and to reduce the amount of required memory, which is a major implementation concern as will be seen in the following. It is important that we stop the SGE when the projected cost of the iterative solver starts to increase again or when memory requirements are small enough so as to fit on the hardware at hand [8].

The Lanczos and Wiedemann algorithms are the most commonly used iterative algorithms in the context of finite fields linear algebra. The Lanczos algorithm is known to have a better complexity than the Wiedemann algorithm. However, the block extension of Wiedemann algorithm (*a.k.a* block Wiedemann) offers the opportunity to split the computation into several independent subtasks, which is an important practical advantage [18,3].

The Wiedemann algorithm and block Wiedemann algorithms return both a vector w of the kernel of A. This vector is non-trivial with high probability. In practice, a single run of the solver is sufficient to find an appropriate solution.

2.1 Wiedemann algorithm

The starting point of the Wiedemann algorithm is to choose two random vectors $x, y \in (\mathbb{Z}/\ell\mathbb{Z})^{N}$. The algorithm is organized in three steps [26], for which we use monikers borrowed from the CADO-NFS software implementation [5].

- The first step computes the first $2N$ terms of the linearly recurrent sequence $(a_i)_{i\in\mathbb{N}} \in (\mathbb{Z}/\ell\mathbb{Z})^{\mathbb{N}}$, where $a_i = {}^t x A^i y$. This step is usually called *Krylov* .
- Then, thanks to the Berlekamp–Massey algorithm, we compute the minimal polynomial of the sequence, which is the polynomial $F(X) = \sum_{i=0}^{d} f_i X^i$ of lowest degree d such that $\sum_{i=0}^{d} f_i a_{k+i} = 0$ for all $k \geq 0$. The degree d is close to N. We commonly call this step *Lingen* .
- The last step, called *Mksol* , finally computes $w = F(A)y$.

The Wiedemann algorithm requires $3N$ SpMVs for the *Krylov* and *Mksol* steps and $O(N \log N)$ field operations for the *Lingen* step. In theory, x should be a random vector of $(\mathbb{Z}/\ell\mathbb{Z})^N$. In practice, we pick a unit vector[1] (*i.e.*, 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.

2.2 Block Wiedemann algorithm

Wiedemann algorithm is fully sequential. In [17,10], Coppersmith et al. presented a block variant that provides parallelism. The block Wiedemann algorithm replaces the vector $y \in (\mathbb{Z}/\ell\mathbb{Z})^N$ by a block of n vectors $y^{(0)}, \ldots, y^{(n-1)}$, each in $(\mathbb{Z}/\ell\mathbb{Z})^N$, and similarly uses a block of m vectors for x. The sequence of scalars a_i is thus replaced by a sequence of m-by-n matrices. There is a complete freedom in the choice of the blocking parameters (m, n). For the efficiency of the *Lingen* run, m is chosen to be equal to $2n$ [5].

- The *Krylov* step now computes the first $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$ terms of the sequence $(a_i)_{i\in\mathbb{N}}$. Notice that the j-th column of the m-by-n matrix ${}^t x A^i y$ depends only on the j-th column of the block vector y. Thus, the computation of $\left({}^t x A^i y\right)_{i\in\mathbb{N}}$ can be distributed into n parallel tasks, each computing $\left({}^t x A^i y^{(j)}\right)_{i\in\mathbb{N}}$. These tasks need no synchronization nor communication, except at the end when all their results are combined.
- The *Lingen* step seeks a linear generator for the previous sequence. The complexity of this step becomes $O\left(n^{\omega-1} N \log N\right)$ with $m = 2n = o(\log N)$ [25]. The output of *Lingen* is composed of n generators $F^{(0)}, \ldots, F^{(n-1)}$, each of them a polynomial over $\mathbb{Z}/\ell\mathbb{Z}$ of degree less than $\lceil \frac{N}{n} \rceil$.
- The *Mksol* step computes the following element of the null-space of A: $w = \sum_{j=1}^{n} F^{(j)}(A) y^{(j)}$. Similarly to the *Krylov* phase, the computation can be distributed into n independent computations.

In the rest of the paper, we focus on the *Krylov* and *Mksol* steps, as they dominate the overall cost and can benefit from parallel hardware. For the *Lingen* computation, we use the CADO-NFS software [5]. Detailed information about this step can be found in [25].

3 The Matrix–Vector Product

The *Lingen* step complexity depends roughly quadratically on the blocking parameter n. Therefore, we can not increase too much the blocking parameters (n, m). We observe also that the block Wiedemann algorithm does not distribute the matrix–vector product, so it does not reduce the amount of required memory per node. Thus, the parallelism provided by the block Wiedemann algorithm is soon limited. We need to explore how to carry out a *Krylov*/*Mksol* task on more than one computation node. Typically, this is related to performing each matrix–vector product in parallel on many computation nodes. In this section, we study how to accelerate this major operation on parallel hardware.

We assume that we have a set of identical *computing nodes* organized according to a 2D rectangular grid and interconnected by a network. Each node is identified by its coordinates (i, j)

[1] For rigorous proofs of success probability, randomization of x is necessary [17]. However, since our input matrix is well-behaved, such randomization is unnecessary.

in the grid. At this level, we ignore the nature of the nodes. The nodes could be cores within a machine, independent machines or GPUs. The matrix A is split into square parts of equal size, such that each node (i, j) gets the part $A_{i,j}$.

3.1 Communication/Computation scheme

An SpMV iteration takes an input vector u and computes $v = Au$. At the beginning of an iteration, a node (i, j) holds the sub-matrix A_{ij} and the j-th fragment u_j of the input vector u. The nodes collaborate together to compute the output vector, which will be the input vector to the next iteration. To be able to run the next iteration, the node A_{ij} only needs to know the j-th fragment v_j of the output vector v. More specifically, the parallel SpMV product is performed as follows.

1. Each node (i, j) computes the partial SpMV $A_{ij}u_j$.
2. Each diagonal node (i, i) collects and sums the partial results from the nodes of the row i. The sum corresponds to the i-th fragment of v.
3. Each diagonal node (i, i) broadcasts its fragment v_j to the nodes of the column i.

In Figure 1, we give an example of a run for 4 parallel nodes with a 2 × 2 split of the matrix. In this figure, the 4 nodes are, represented in gray, numbered from 0 to 3. On the left-hand side, we indicate how the matrix A and the input vector u are distributed among the nodes. We detail on the right-hand side the intermediate data present on each node after each step.

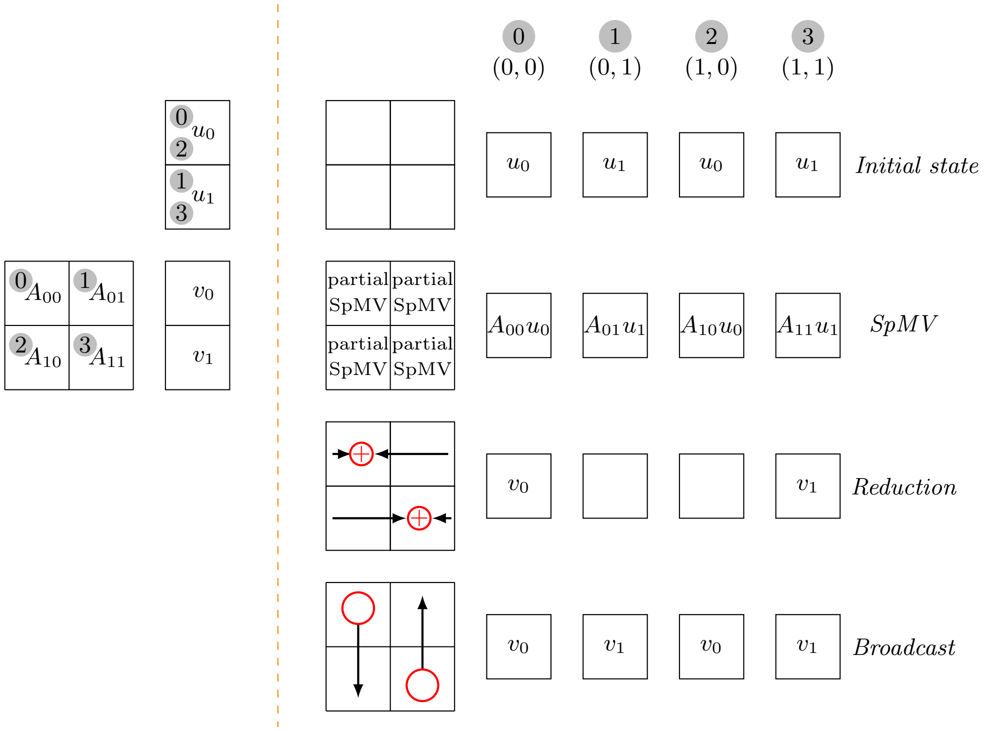

Fig. 1: Computation/Communication scheme for a 2 × 2 split of A

The communication scheme suffers from the fact that only one node per row collects the partial products. A parallelization of the Reduction/Broadcast operations is possible, typically using the ReduceScatter/AllGather operations. This should yield to a significant speedup of the communication delay. However, the output of the iteration will be permuted, *i.e.,* the fragments of v will not be distributed as were those of u in the beginning of the iteration. In summary, it remains an improvement that can be explored.

3.2 Balancing the workload

The particular distribution of the non-zero coefficients is such that the nodes will get unbalanced workloads, and the nodes working on the denser parts will take more time than those working on the sparser ones.

For the particular kind of input, this unbalance problem can fortunately be solved efficiently. To fix this problem, we apply permutations of the rows and columns, so that the distribution of non-zero coefficients for each sub-matrix is close to that of the matrix A, as shown in Figure 2. One possibility to obtain this permutation is to sort the columns by their weight and distribute them evenly among the nodes, then proceed likewise with the rows. This is made possible by the fact that the standard deviation of the row weight is much smaller than that of the column weight.

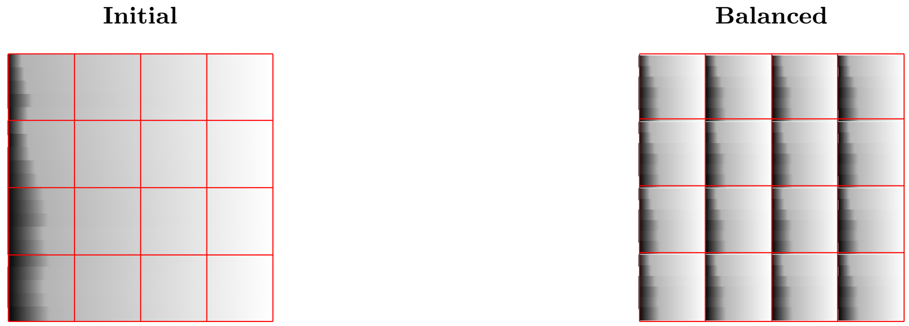

Fig. 2: Distribution of non-zero coefficients for initial and balanced matrices

3.3 The partial SpMV

The matrix is stored in a sparse format, adapted from the *Compressed Sparse Row* (CSR) format for the particular distribution of the non-zero coefficients.

We chose to implement the arithmetic operations in $\mathbb{Z}/\ell\mathbb{Z}$ using the *Residue Number System* (RNS). The use of this representation for finite field arithmetic provides a fine grained parallelism, which can be exploited by Single Instruction, Multiple Data (SIMD) architectures.

In the remainder of the article, we consider the partial matrix–vector product as a *black box*, that is, a subroutine which, on inputs A and u returns the product Au. We give more details about how this subroutine is implemented in [14].

3.4 Communication concerns

We now focus on how to share data between the computing nodes, in the cases of CPU nodes and GPU nodes.

CPU communications The case of CPU-only setups is quite straightforward, as we use the MPI operation `MPI_Reduce` to collect and combine on a diagonal node the results of nodes belonging to the same row, and `MPI_Bcast` to broadcast the combined results to the nodes of each column.

In the following subsections, we assume that we execute the application over a cluster of GPUs and we discuss the data movement. We restrict to NVIDIA graphics hardware. Distributing an SpMV on several GPUs requires considering two possible (and not mutually exclusive) cases: the first one where a single CPU node harbors two or more GPUs, and the second one where the GPUs are in different CPU nodes.

Intra-node GPU communications We are in the case of sharing data between two GPUs within the same CPU node. In order to do so, *CUDA*, the parallel programming model for NVIDIA GPUs [1] offers three possibilities:

- Staging through CPU: the communication has to involve the host CPU. Thus, it is composed of two transfers, a device-to-host copy (D2H) then a host-to-device copy (H2D).
- Device-to-device copy (D2D): from the programmer's perspective, it is a direct copy of the GPU buffers. Although the transfer still passes through the host memory, the copy is fully pipelined.
- Peer-to-Peer Direct Access (P2P DMA): using this feature, the devices can share data independently of the CPU. P2P DMA requires to enable peer access for each GPU, which is supported by recent hardware.

The P2P DMA feature should decrease the host overhead and thus accelerate the memory copies. To verify it, we ran benchmarks to compare the bandwidth and latency of each approach (cf. Figure 3). The experiment is performed using two NVIDIA GeForce GTX 680 cards. The benchmarks measure the run time for sending messages of increasing size from one GPU to the other. The latencies for the first two options are 19.7 µs and 19.4 µs, respectively, and only 14 µs when the P2P DMA is enabled. The peak bandwidths are 6.1 GB/s for the explicit host staging transfer, 7.3 GB/s for the device to device transfer, and 10.4 GB/s for the P2P DMA transfer.

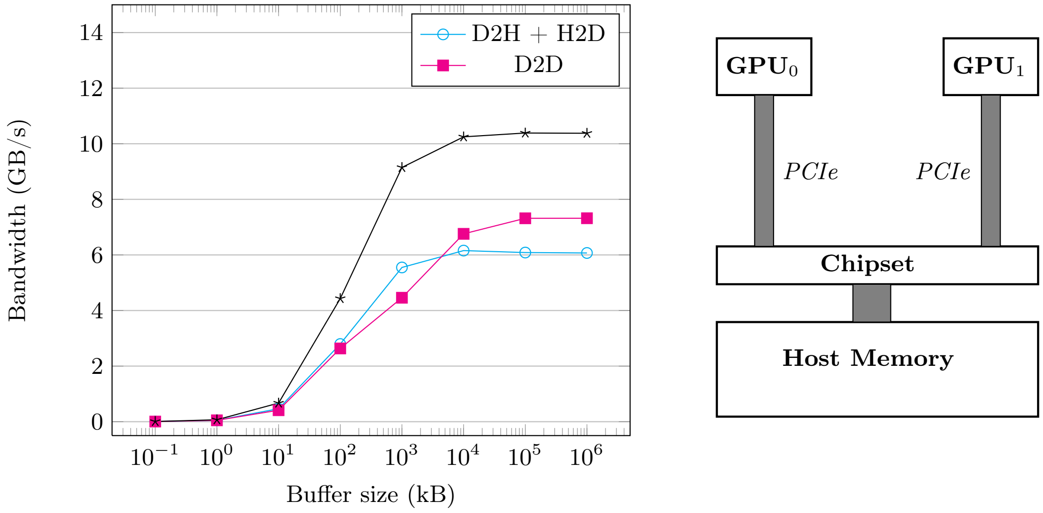

Fig. 3: Benchmarking Intra-node GPU communications

Inter-node GPU communications Now, we are interested in the case of sharing data between GPUs installed in different CPU nodes. The trivial option in this case is to perform the transfer in three steps: a data copy from device to host using CUDA routines, then use MPI to copy data between hosts, and finally a CUDA copy from host to device on the destination node (cf. Figure 4).

It is however possible to overcome the host staging using the *Cuda-aware MPI* feature which combines MPI and CUDA. It allows one to address GPU buffers directly in the MPI routines (cf. Figure 5). From the programmer's point of view, a data transfer boils down to one call to an MPI routine. With *Cuda-aware MPI*, the data transfers are fully pipelined, while without the feature, the transfers between hosts and those between the device and the host are pipelined separately.

The *Cuda-aware MPI* feature is incorporated in several widely used MPI libraries and considerably improves the data movement latencies.

Another feature that further optimizes data transfers is *GPU Direct*. It allows one to move data directly from the GPU buffers to the IB buffers (cf. Figure 6). We could not deploy this feature in our application, as it is supported only by the recent Tesla and Quadro cards.

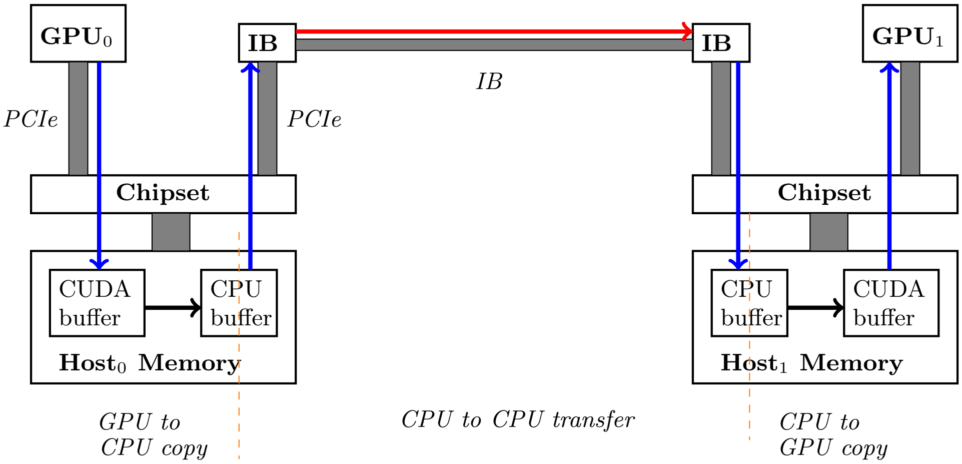

Fig. 4: Data copy from GPU_0 to GPU_1 without *Cuda-aware MPI*

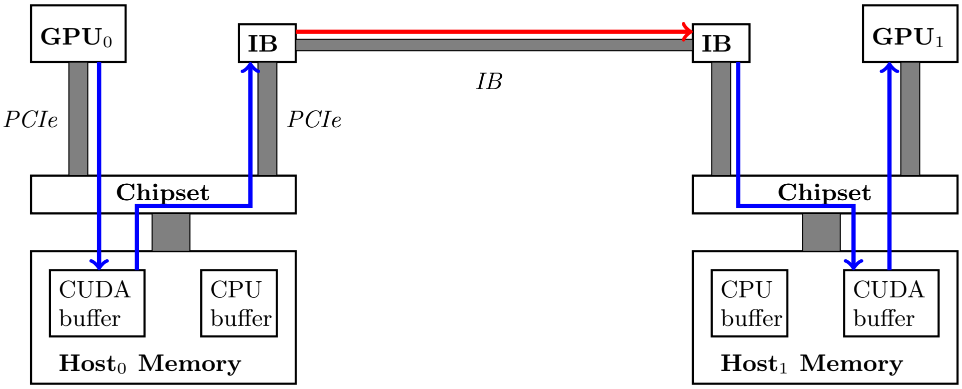

Fig. 5: Data copy from GPU_0 to GPU_1 with *Cuda-aware MPI*

In Figure 7, we report the results of bandwidth benchmarks for inter-node GPU-to-GPU communications. We ran the experiment using two NVIDIA GTX 680 installed in two nodes connected with QDR InfiniBand. We use CUDA 5.0 and Open MPI 1.7.3. In addition to benchmarks for the two ways of communication, we added the Host-to-Host (H2H) communication results as a reference, for which we measured the data movement from one CPU buffer to another CPU buffer using the regular MPI routines.

The latency of a plain Device-to-Device transfer is 11 µs. It becomes 9 µs if the feature *Cuda-aware MPI* is used. The latency of the Host-to-Host transfer is 1 µs. Without *Cuda-aware MPI*, the bandwidth is bounded by 2.3 GB/s. The *Cuda-aware MPI* feature allows to reach the Host-to-Host peak bandwidth, which is 3.7 GB/s.

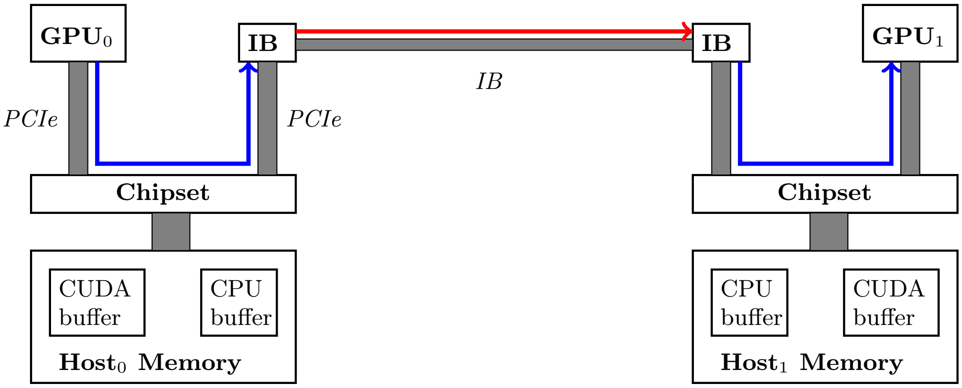

Fig. 6: Data copy from GPU$_0$ to GPU$_1$ with *Cuda-aware MPI* and *GPU Direct*

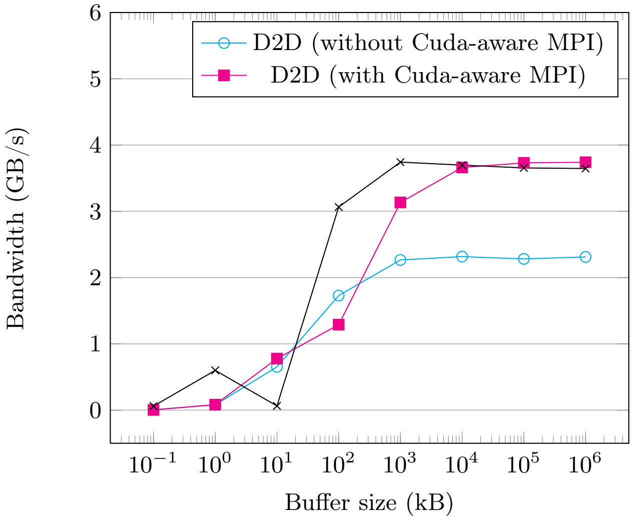

Fig. 7: Benchmarking Inter-node GPU communications

4 Examples of computations

4.1 DLP in GF(2^{809})$^\times$ using FFS

The function field sieve (FFS) [2] is an *index-calculus* algorithm designed to attack the DLP in the multiplicative subgroup of a finite field GF(p^n), where the characteristic p is a small prime. Barbulescu et al. announced in [6] the solving of the DLP in the 202-bit prime order subgroup of GF(2^{809})$^\times$ using FFS. This computation is the largest DLP computation in a binary field extension of prime degree. The previous record was the computation of a DLP in GF(2^{613})$^\times$ [16].

Matrix. In this computation, the linear algebra step is performed in $\mathbb{Z}/\ell\mathbb{Z}$ where ℓ is 202 bits long. The relation collection phase produced an initial matrix of 78.8M rows. A preliminary structured Gaussian elimination reduced the matrix to 3,602,667 rows and columns, with an average of 100 non-zero coefficients per row. Each non-zero coefficient of A fits in a single machine word. Around 90% of them are ±1 [8], [6].

Linear Algebra Setup. At the time of the computation, we had access to a 4-node cluster, with 2.4 GHz Intel Xeon E5620 Westmere processors connected with InfiniBand network at 40 Gb/s. Each node is equipped with 2 NVIDIA Tesla M2050 graphics processors[2].

The total memory required to handle the matrix along with the input and output vectors is 3.16 GB. Since the available memory on one card is only 3 GB, the block Widemann configuration $(n = 8, m = 16)$, for which a sequence $\left({}^t x A^i y^{(j)}\right)_{i \in \mathbb{N}}$ can be computed on a single device, is not feasible. We have to compute each sequence on more than one device; the configuration $(n = 4, m = 8)$ with a 2×1 split of the matrix and the configuration $(n = 2, m = 4)$ with a 2×2 split of the matrix are possible. Theoretically, the former appears to be the most convenient, since only two GPUs connected to the same node communicate, while, with the latter, 4 GPUs interconnected with the network are required to communicate.

In the following table, we detail a comparison between these two configurations. The comparison shows how the inter-node GPU communication for the second configuration slows down the overall computation time. We also present benchmarks related to a smaller matrix, for which the three configurations are possible and a bigger matrix, for which only the $(n = 2, m = 4)$ configuration is feasible.

Matrix size (required memory)	Possible blocking parameters	SpMV + comm. delays per iteration	Overall comp. time	Ratio comm./iteration
3.6M × 3.6M (3.2 GB)	$\boldsymbol{(n = 4, m = 8)}$	**142 + 27 ms**	**4.5 days**	**16%**
	$(n = 2, m = 4)$	72 + 41 ms	6 days	37%
3M × 3M (2.7 GB)	$\boldsymbol{(n = 8, m = 16)}$	**228 + 0 ms**	**2.5 days**	**0%**
	$(n = 4, m = 8)$	115 + 23 ms	3 days	17%
	$(n = 2, m = 4)$	58 + 35 ms	4.1 days	38%
6M × 6M (5.4 GB)	$\boldsymbol{(n = 2, m = 4)}$	**123 + 69 ms**	**16.7 days**	**36%**

With the $(n = 4, m = 8)$ blocking parameters, an iteration takes 169 ms on each node, including 27 ms for the GPU communications. The initial sequence computation required 2.6 days in parallel on the 4 nodes. The linear generator computation was carried out in parallel using 16 threads running on 16 CPU cores. It required 2 hours. Finally, computing the kernel vector required 1.8 days in parallel on the 4 GPU nodes. The overall computation took a total wall-clock time of about 4.5 days.

4.2 DLP in a 596-bit prime field using NFS

To compute discrete logarithms in a prime field GF(p), the Number Field Sieve (NFS) algorithm is used [15]. The last NFS record was accomplished by T. Kleinjung et al. [4] for a 530-bit (160-decimal-digit) prime p using NFS. We are currently running an NFS-based computation to attack the DLP in a 596-bit (180-decimal-digit) prime field. The linear algebra step is defined over a 595-bit prime ℓ.

Matrix. The matrix contains 179M rows at the end of the relation collection. The preliminary structured Gaussian elimination reduced the number of rows to 7,287,476, with an average weight of 150 non-zero coefficients per row. The matrices issued from NFS computations contain a small number (5, here) of dense columns, whose elements live in $\mathbb{Z}/\ell\mathbb{Z}$. The rest of the matrix is similar to an FFS matrix in terms of distribution and coefficient size. Taking this dense part into account adds a non-negligible cost when compared to FFS matrices.

[2] This work was realized with the support of HPC@LR, a Center of Competence in High-Performance Computing from the Languedoc-Roussillon region, funded by the Languedoc-Roussillon region, the European Union and the Université Montpellier 2 Sciences et Techniques. The HPC@LR Center is equipped with an IBM hybrid Supercomputer.

GPU Setup. For this computation, we have access to 8 NVIDIA GeForce GTX 680 graphics processors, plugged into a 4-node cluster of Intel Xeon E5-2609 processors running at 2.4 GHz, and connected with QDR Infiniband network. Each graphics card has 4 GB of memory.

The total memory required to carry out the SpMV on one GPU is 9.8 GB. Thus, 4 GPUs should work on a single sequence, *i.e.*, at most two sequences can be computed in parallel, and the blocking parameters are $(n = 2, m = 4)$.

An iteration takes 615 ms on each group of 4 GPUs, with 195 ms for the GPU-to-GPU communications. The overall computation should take a total wall-clock time of about 65 days.

CPU Setup. Another option was tried, using our CPU implementation on a 768-core cluster. The cluster contains 48 nodes connected with FDR Infiniband. Each node hosts two 2-GHZ 8-core Intel Xeon E5-2650 processors.

With this setup, we propose an 8×8 split of the matrix, so that 64 MPI processes running on 4 nodes work together to carry out a matrix–vector product, each process running on one core. This yields to a $(n = 12, m = 24)$ block Wiedemann configuration. The processes are distributed so that the processes running on the same node are contiguous (cf. Figure 8). This allows to accelerate the reduction/broadcast operations, since data sharing between threads belonging to the same node is performed on shared memory, not across the network. A speedup of 2.4 on the communication delay is observed when comparing with the default MPI processes mapping.

In the following table, we compare the GPU and CPU setups. We observe that starting from a certain matrix size and with these setups, the multi-core acceleration prevails over the GPU one. For comparison, we add an hypothetical setup, where we have a cluster similar to the 48-node multi-core cluster, but containing two NVIDIA GeForce GTX 680 on each node.

Matrix size (Memory)	Setup	Blocking parameters	SpMV + comm. delays [ms]	Overall comp. time	Ratio comm./iteration
7.3M × 7.3M (9.8 GB)	8 GPUs on 4 nodes	$(n = 2, m = 4)$ 4 GPUs ↔ 1 subtask	420 + 195	65 days	32%
	768 cores on 48 nodes	$\boldsymbol{(n = 12, m = 24)}$ **64 cores ↔ 1 subtask**	**1700 + 400**	**50 days**	**19%**
	96 GPUs on 48 nodes	$(n = 24, m = 48)$ 4 GPUs ↔ 1 subtask	420 + 195	5.5 days	32%

With the CPU setup, an iteration is performed in 2.1 s by the 64 parallel threads, including 0.4 ms for communications. The *Krylov* phase required 28 days by using in average 75% of the cluster resources (*i.e.*, 576 cores), which is equivalent to 45-core years. The *Lingen* phase requires 10 days, and the *Mksol* phase should take around 20 days.

5 Conclusion

In this article, we presented how the block solvers, in our case block Wiedemann algorithm, distribute heavy computations without an additional overhead. We discussed a further parallelization of the matrix-vector product and detailed how we can efficiently run this computation in a cluster of GPUs or CPUs. In the examples that we ran, we did not combine the two architectures on the same computation. However, our final implementation can be run on a hybrid GPU/Multi-core architecture.

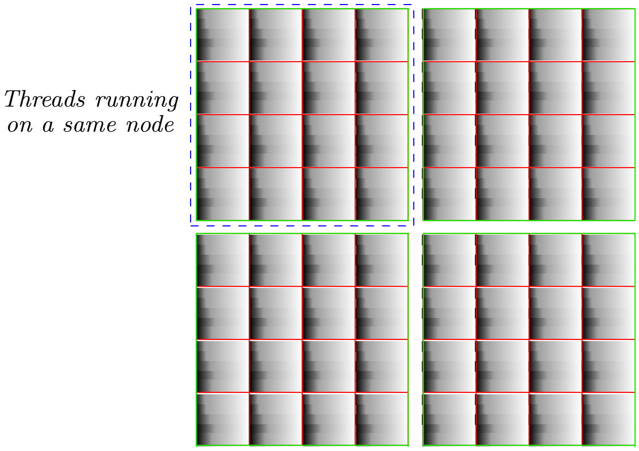

Fig. 8: 8 × 8 split of the matrix and mapping on four 16-core nodes

References

1. *CUDA Programming Guide Version 5.5*, 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.
2. L. M. Adleman. The function field sieve. In L. M. Adleman and M.-D. A. Huang, editors, *ANTS-I*, volume 877 of *Lecture Notes in Comput. Sci.*, pages 108–121. Springer–Verlag, 1994. Proceedings.
3. K. Aoki, J. Franke, T. Kleinjung, A. Lenstra, and D. A. Osvik. A kilobit special number field sieve factorization. Cryptology ePrint Archive, Report 2007/205, 2007. `http://eprint.iacr.org/`.
4. F. Bahr, J. Franke, and T. Kleinjung. Discrete logarithms in GF(p) - 160 digits, February 2007. Email to the NMBRTHRY mailing list. Available at `http://perso.univ-rennes1.fr/reynald.lercier/file/BFK07.txt`.
5. S. Bai, A. Filbois, P. Gaudry, A. Kruppa, F. Morain, E. Thomé, and P. Zimmermann. CADO-NFS, Crible Algébrique: Distribution, Optimisation - Number Field Sieve. `http://cado-nfs.gforge.inria.fr/`.
6. 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 Hugo Krawczyk, editor, *PKC 2014*, Buenos Aires, Argentina, 2014. Springer.
7. R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. EUROCRYPT 2014.
8. C. Bouvier. The filtering step of discrete logarithm and integer factorization algorithms. Preprint, 22 pages, available at `http://hal.inria.fr/hal-00734654`, 2013.
9. B. Boyer, J. G. Dumas, and P. Giorgi. Exact sparse matrix-vector multiplication on GPU's and multicore architectures. *CoRR*, abs/1004.3719, 2010.
10. D. Coppersmith. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.*, 62(205):333–350, January 1994.
11. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
12. R. Dutta, R. Barua, and P. Sarkar. Pairing-based cryptographic protocols : a survey. Cryptology ePrint Archive, Report 2004/064, 2004. `http://eprint.iacr.org/`.
13. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
14. H. Jeljeli. Accelerating iterative SpMV for Discrete Logarithm Problem using GPUs. Preprint, 11 pages, available at `http://hal.inria.fr/hal-00734975`, 2013.
15. A. Joux and R. Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. a comparison with the Gaussian integer method. *Mathematics of computation*, 72(242):953–967, 2003.
16. A. Joux and R. Lercier. Discrete logarithms in GF(2^{607}) and GF(2^{613}). E-mail to the NMBRTHRY mailing list; `http://listserv.nodak.edu/archives/nmbrthry.html`, September 2005.
17. E. Kaltofen. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Mathematics of Computation*, 64(210):777–806, 1995.

18. T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010. http://eprint.iacr.org/.
19. C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.
20. A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance, 1984.
21. C. Pomerance and J.W. Smith. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experiment. Math*, 1:89–94, 1992.
22. J. Zumbragel R. Granger, T. Kleinjung. Discrete logarithms in GF(2^{9234}). E-mail to the NMBRTHRY mailing list; http://listserv.nodak.edu/archives/nmbrthry.html, January 2014.
23. B. Schmidt, H. Aribowo, and H-V. Dang. Iterative sparse matrix-vector multiplication for integer factorization on GPUs. 6853:413–424, 2011.
24. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
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. D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theor.*, 32(1):54–62, January 1986.