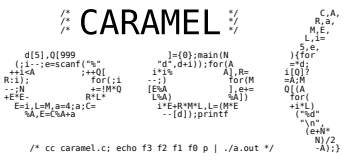
Resolution of Linear Algebra for the Discrete Logarithm Problem Using GPU and Multi-core Architectures

Hamza Jeljeli

CARAMEL project-team, LORIA, INRIA / CNRS / Université de Lorraine,
Hamza.Jeljeli@loria.fr

Euro-Par 2014, Porto, August 27th, 2014



Discrete Logarithm Problem

- Inverse operation of the exponentiation (in a cyclic group).
- In certain groups, the discrete logarithm problem (DLP) is computationally hard.
- The inverse problem (discrete exponentiation) is easy.
- Security of **cryptographic primitives** relies on the difficulty of DLP:
  - Key agreement: Diffie–Hellman key exchange,
  - Encryption: ElGamal encryption,
  - Signature: DSA signature,
  - Pairing-based cryptography, ...

- Evaluate security level of primitives $\Longrightarrow$ DLP attacks.

Linear Algebra Issued from DLP Attacks

- A particular interest in DLP in **finite fields GF(q)**.
- To attack DLP in finite fields, *index-calculus* methods:
  - solve DLP in time sub-exponential or quasi-polynomial in the size of the finite field.
  - require solving large sparse systems of linear equations over finite fields.
- Inputs:
  - a prime ℓ that divides $q - 1$,
  - a matrix A.
- Output: a non-trivial vector w such that

$$Aw \bmod \ell = 0.$$

Characteristics of the Inputs

- ℓ between 100 and 1000 bits.
- A is an N-by-N sparse matrix.
- 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.

- Non-zero coefficients in $\mathbb{Z}/\ell\mathbb{Z}$: majority are "small" (32 bit integers).

Related Issues to Linear Algebra

- Exact computations: unique solution,
- Matrix size $\Longrightarrow$ storage issues,
- To solve linear algebra, need to parallelize:
  - algorithmic level: block algorithms,
  - matrix-vector product level in many computing nodes,
  - per-node computation,
  - and arithmetic over $\mathbb{Z}/\ell\mathbb{Z}$: use of RNS.

  } talk at WAIFI 2014.

$\Rightarrow$ Use of clusters of **multi-core CPUs** and/or **GPUs**.

- Communication concerns,

$\Rightarrow$ Nodes interconnected by fast communication links (**InfiniBand**).

Motivation

- Optimize the use of High-Performance Computing resources to solve linear algebra efficiently.
- Minimize the **overall wall-clock time** for solving the problem.
- Run DLP computations.

Table of Contents

Table of Contents

Direct and Iterative Methods

To solve systems of linear equations, 2 families of algorithms:

- Direct methods: Gaussian elimination or LU/QR decompositions
- Iterative methods: Conjugate gradient method and Lanczos and Wiedemann algorithms.

Direct methods	**Iterative methods**
$O\left(N^{\omega}\right)$ field operations ($\omega = 2.81$ using Strassen algorithm)	$O\left(N\right)$ sparse-matrix–vector products (SpMVs)
densify the matrix $\Rightarrow$ storage issues	do not modify the matrix

- As long as cost(SpMV) $< O\left(N^{\omega-1}\right)$ field operations, iterative methods are **asymptotically faster**.
- Cost(SpMV) $= O\left(N\gamma\right)$, where γ is the average number of non-zero coefficients per row ($\gamma \ll N$).

Lanczos Algorithm or Wiedemann Algorithm?

- Lanczos algorithm has a **better complexity** than Wiedemann algorithm.
- However, the block extension of Wiedemann algorithm offers the opportunity to **split the computation into several independent subtasks**.

Wiedemann Algorithm [Wiedemann 1986]

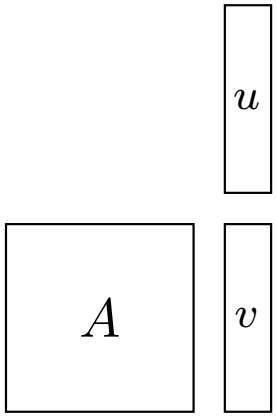
0. Take 2 random vectors $x, y \in (\mathbb{Z}/\ell\mathbb{Z})^N$
1. Krylov: compute first $2N$ terms of $a_i = {}^t x A^i y \implies \mathbf{2N}$ **SpMVs**.
2. Lingen: compute minimal polynomial of the a_i's,
   Output: polynomial F of degree d, close to N.
   $\implies \boldsymbol{O\left(N(\log N)^2\right)}$ **field operations**
3. Mksol: compute $w = F(A)y$. $\implies \boldsymbol{N}$ **SpMVs**

Overall complexity dominated by Krylov and Mksol:
$\mathbf{3N}$ **SpMVs** $\implies \boldsymbol{O\left(N^2\gamma\right)}$ **field operations**.

Block Wiedemann Algorithm [Coppersmith 1994]

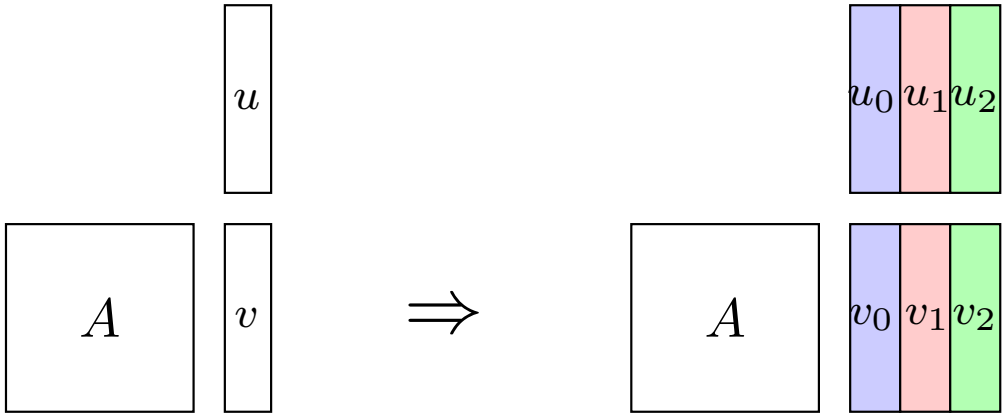
0. Replace vectors x, y by blocks of n vectors
1. Krylov: compute first $\left\lceil \frac{2N}{n} \right\rceil$ terms of $a_i \implies \left\lceil \frac{\mathbf{2N}}{\boldsymbol{n}} \right\rceil$ **SpMbVs**
2. Lingen: compute minimal polynomial of the a_i's,
   Output: n polynomials F_j, each of degree d, close to $\left\lceil \frac{N}{n} \right\rceil$.
   $\implies \boldsymbol{O\left(n^{\omega-1} N (\log N)^2\right)}$ **field operations**
3. Mksol: compute $w = \sum_{j=1}^{n} F_j(A) y_j \implies \left\lceil \frac{\boldsymbol{N}}{\boldsymbol{n}} \right\rceil$ **SpMbVs**.

Parallelism Provided by BW



SpMV
with Wiedemann

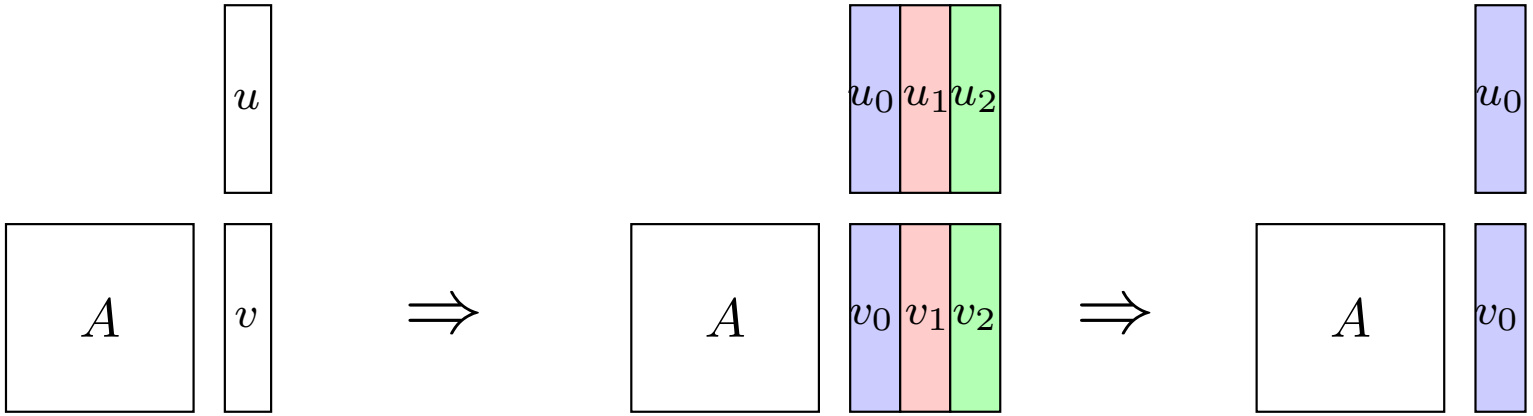
Parallelism Provided by BW



SpMV
with Wiedemann

SpMbV
with Block Wiedemann

Parallelism Provided by BW



SpMV
with Wiedemann

SpMbV
with Block Wiedemann

Parallelism Provided by BW

1. Divide the number of Krylov/Mksol iterations by n.
   $\Rightarrow$ Improve the overall complexity (SpMbV costs less than n SpMVs).
2. Distribute Krylov/Mksol computation **into n parallel tasks**.
   $\Rightarrow$ No synchronization nor communication, except at the end when all their results are combined.

Parallelism Provided by BW

1. Divide the number of Krylov/Mksol iterations by n.
   $\Rightarrow$ Improve the overall complexity (SpMbV costs less than n SpMVs).
2. Distribute Krylov/Mksol computation **into n parallel tasks**.
   $\Rightarrow$ No synchronization nor communication, except at the end when all their results are combined.

However, BW parallelism is soon limited:

- Lingen complexity depends roughly quadratically on n.
  $\Rightarrow$ We can not increase too much the blocking parameter n.
- BW algorithm does not distribute the matrix–vector product, so it does not reduce required memory per node.

$\Rightarrow$ Need to explore how to carry out a Krylov/Mksol task (matrix-vector product) **in parallel in many computing nodes**.

Table of Contents

The Model

- A set of identical computing nodes organized according to a **2D square grid** and interconnected by a **network**.
- A node could be a core within a machine, an independent machine or a GPU.
- Each node identified by its coordinates (i, j) in the grid.
- The matrix A is split into square parts of equal size, so that each node (i, j) gets the part $A_{i,j}$.

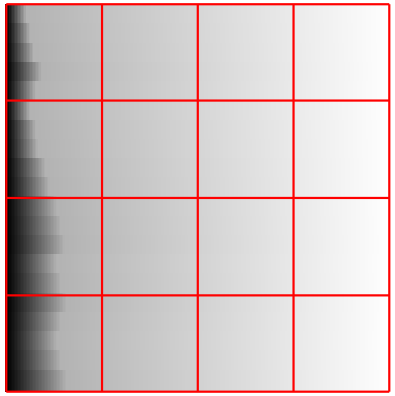
Objective

Given the matrix and an input vector u, the nodes collaborate together to compute the output vector $v = Au$.

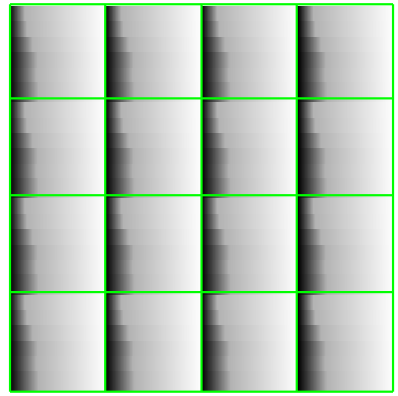
Balancing Workload

- Trivial split $\Longrightarrow$ nodes will get unbalanced workloads (due to sparsity unbalance of A).
- We apply permutations of rows and columns, so that distribution (each sub-matrix) $\approx$ distribution (A).
- One possibility to obtain this permutation :
  - sort columns (by weight), distribute evenly
  - proceed likewise with the rows.

Trivial

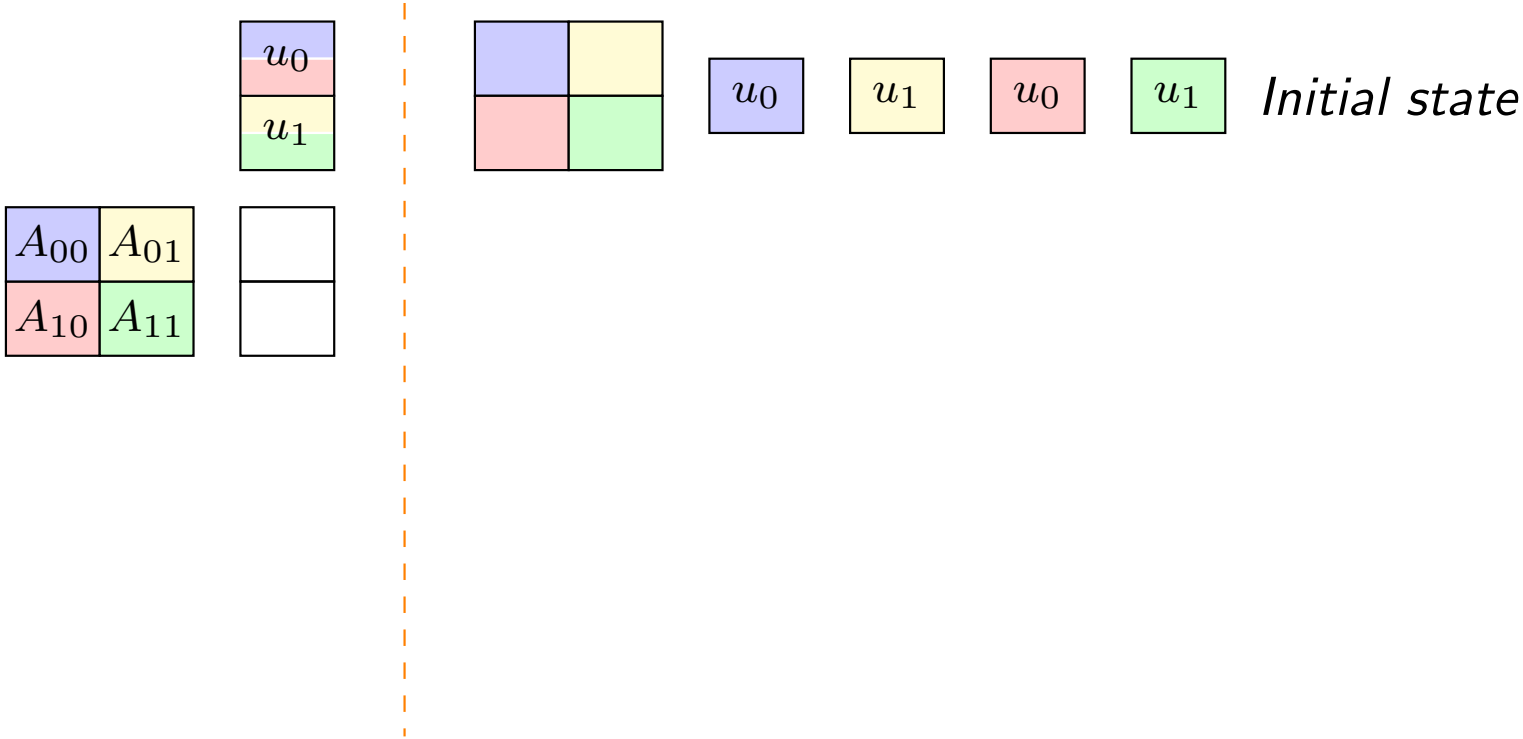


Balanced



Communication/Computation Scheme

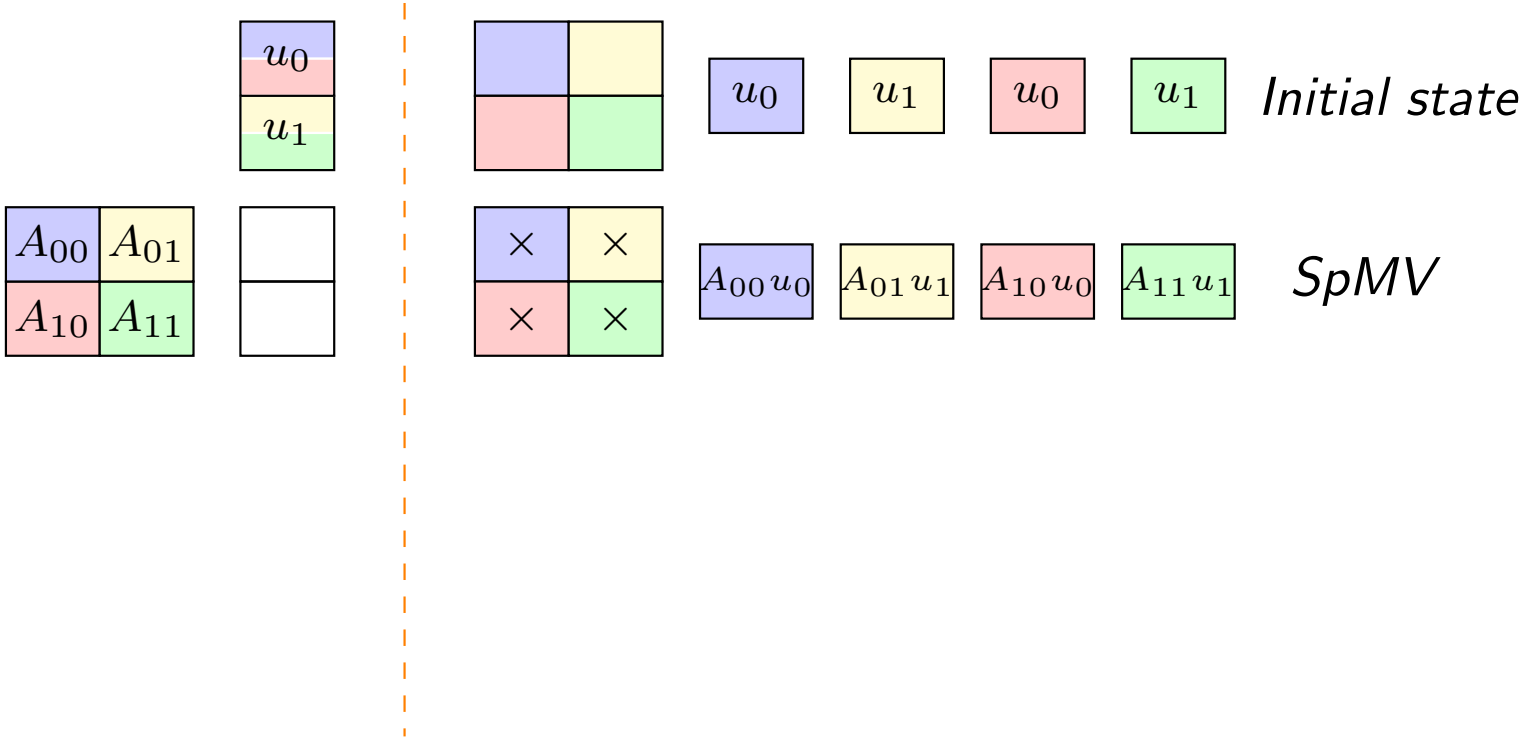
Beginning of iteration: node (i, j) holds A_{ij} and j-th fragment u_j of u.
Next iteration: node (i, j) only needs to know j-th fragment v_j of v.



Communication/Computation Scheme

Beginning of iteration: node (i, j) holds A_{ij} and j-th fragment u_j of u.
Next iteration: node (i, j) only needs to know j-th fragment v_j of v.

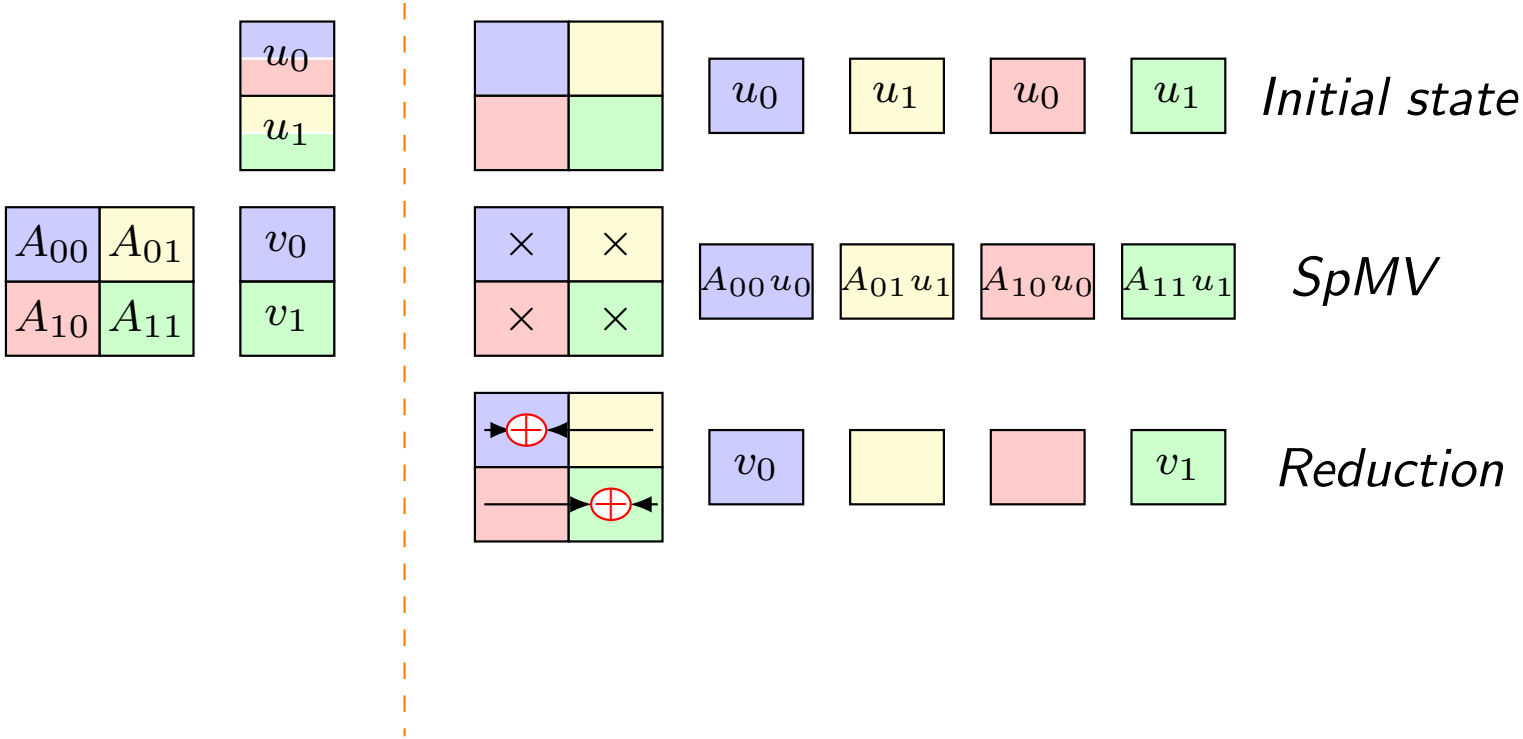
1. Each node (i, j) computes partial SpMV $A_{ij}u_j$.



Communication/Computation Scheme

Beginning of iteration: node (i, j) holds A_{ij} and j-th fragment u_j of u.
Next iteration: node (i, j) only needs to know j-th fragment v_j of v.

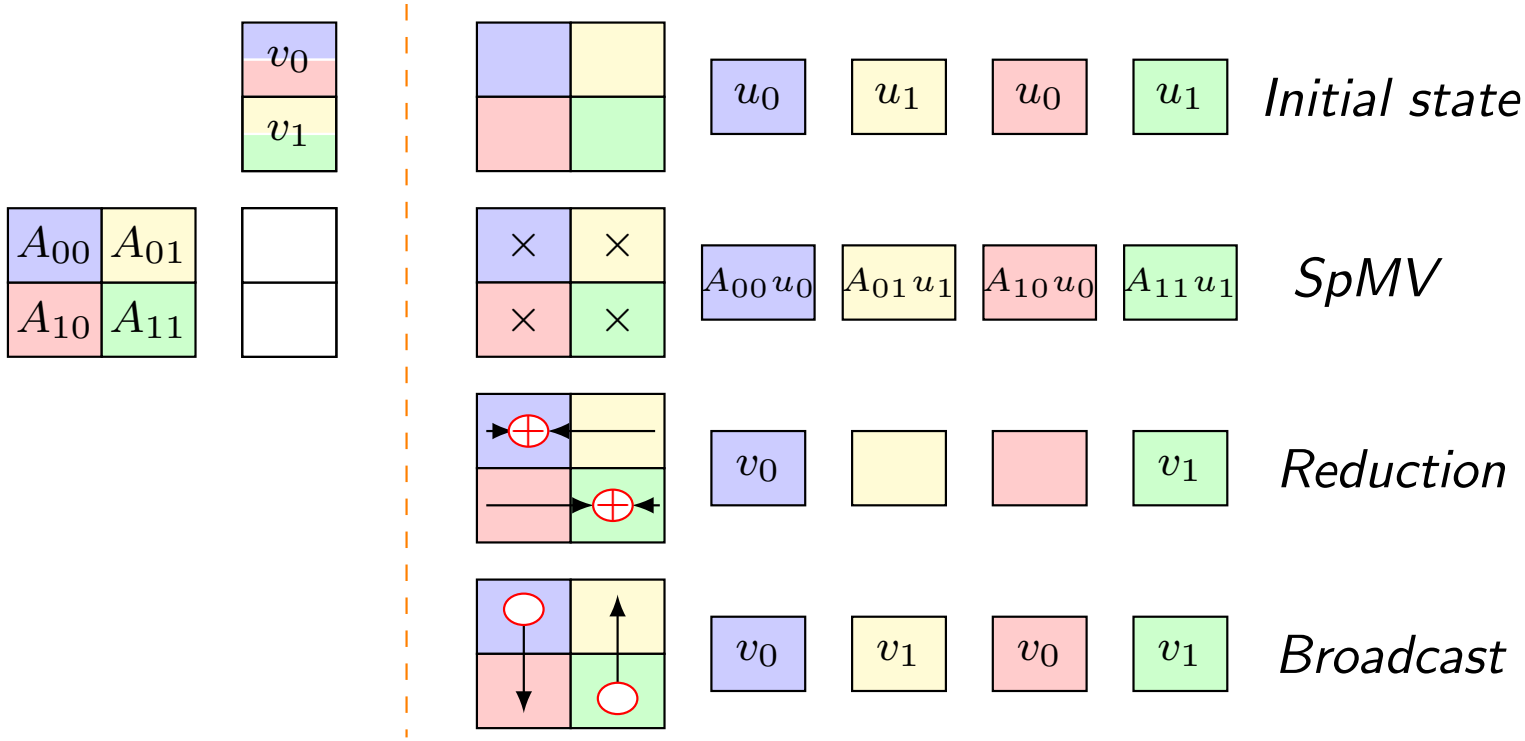
1. Each node (i, j) computes partial SpMV $A_{ij}u_j$.
2. Each diagonal node (i, i) collects and sums partial results from nodes of row i. The sum corresponds to i-th fragment of v.



Communication/Computation Scheme

Beginning of iteration: node (i, j) holds A_{ij} and j-th fragment u_j of u.
Next iteration: node (i, j) only needs to know j-th fragment v_j of v.

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



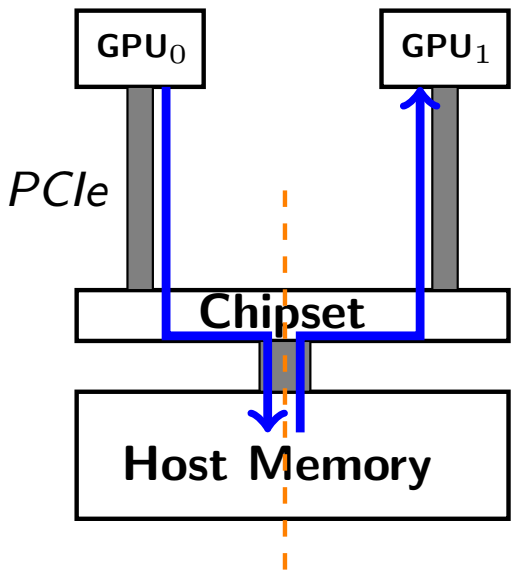
Communications: Share Data Between Computing Nodes

- CPU setups: MPI operations:
  - `MPI_Reduce`: collect and combine results of nodes belonging to the same row.
  - `MPI_Bcast`: broadcast combined results to nodes of each column.
- NVIDIA GPU setups: consider 2 cases:
  - Intra-node: a single CPU node harbors two or more GPUs.
  - Inter-node: GPUs are in different CPU nodes

Intra-node GPU Communications

CUDA offers 3 possibilities:

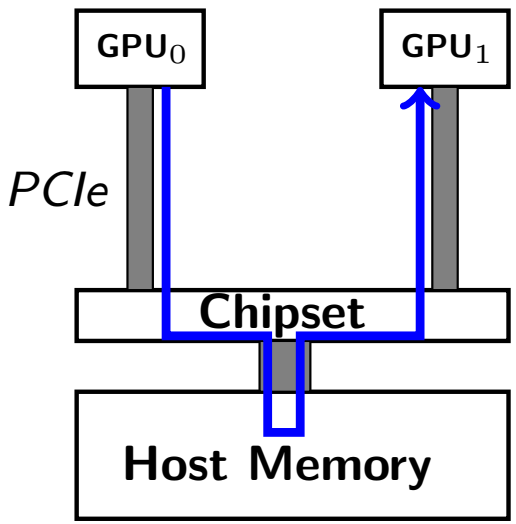
1. Staging through CPU: 2 transfers,
   - a device-to-host copy (D2H),
   - then, a host-to-device copy (H2D).



Intra-node GPU Communications

CUDA offers 3 possibilities:

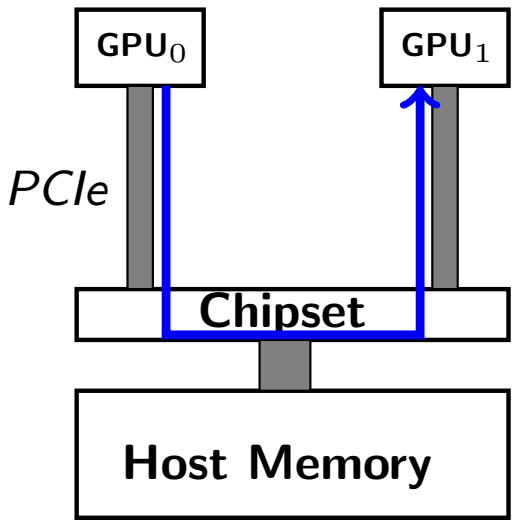
1. Staging through CPU: 2 transfers,
   - a device-to-host copy (D2H),
   - then, a host-to-device copy (H2D).
2. Device-to-device copy (D2D): transfer still passes through host memory, copy is fully pipelined.



Intra-node GPU Communications

CUDA offers 3 possibilities:

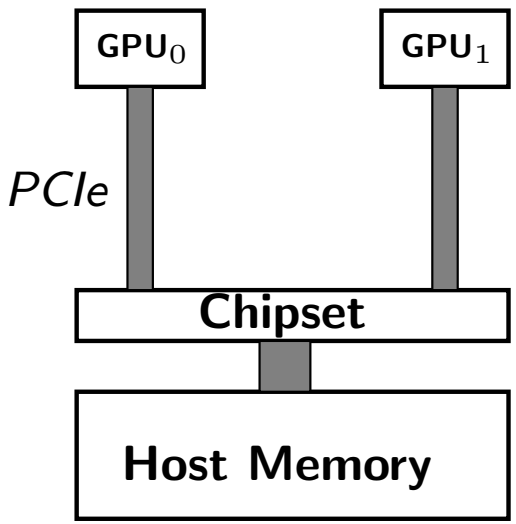
1. Staging through CPU: 2 transfers,
   - a device-to-host copy (D2H),
   - then, a host-to-device copy (H2D).
2. Device-to-device copy (D2D): transfer still passes through host memory, copy is fully pipelined.
3. Peer-to-Peer Direct Access (P2P DMA): devices can share data independently of CPU.

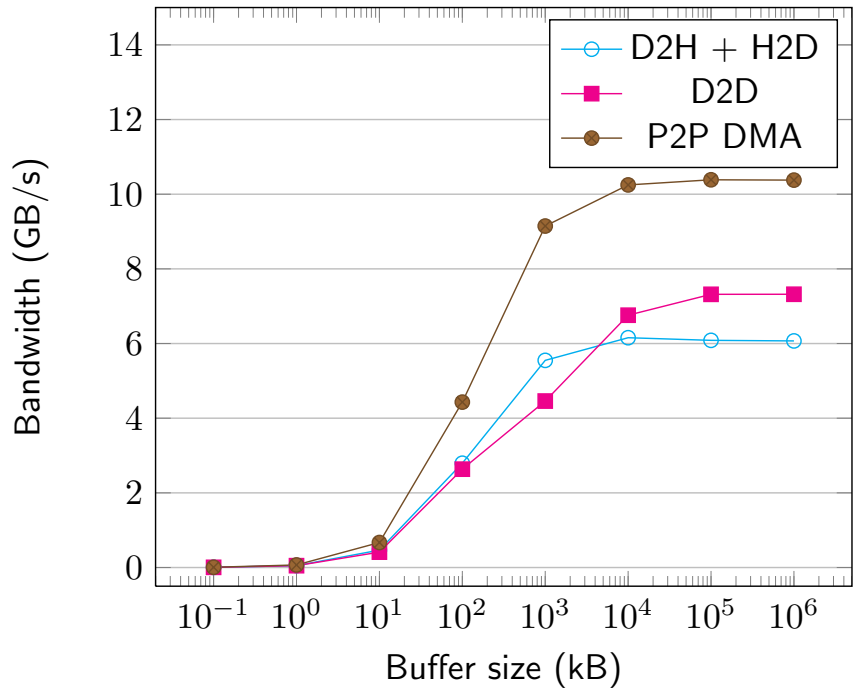


Intra-node GPU Communications

CUDA offers 3 possibilities:

1. Staging through CPU: 2 transfers,
   - a device-to-host copy (D2H),
   - then, a host-to-device copy (H2D).
2. Device-to-device copy (D2D): transfer still passes through host memory, copy is fully pipelined.
3. Peer-to-Peer Direct Access (P2P DMA): devices can share data independently of CPU.

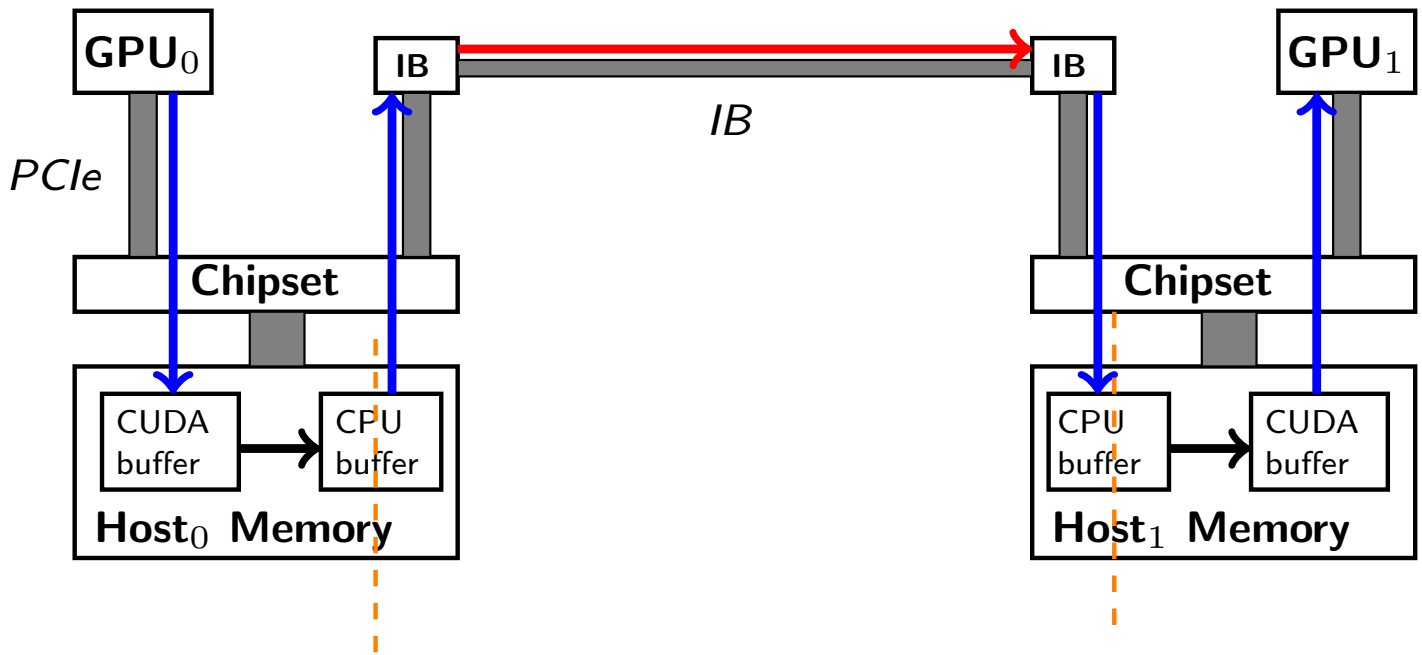




Inter-node GPU Communications

2 options:

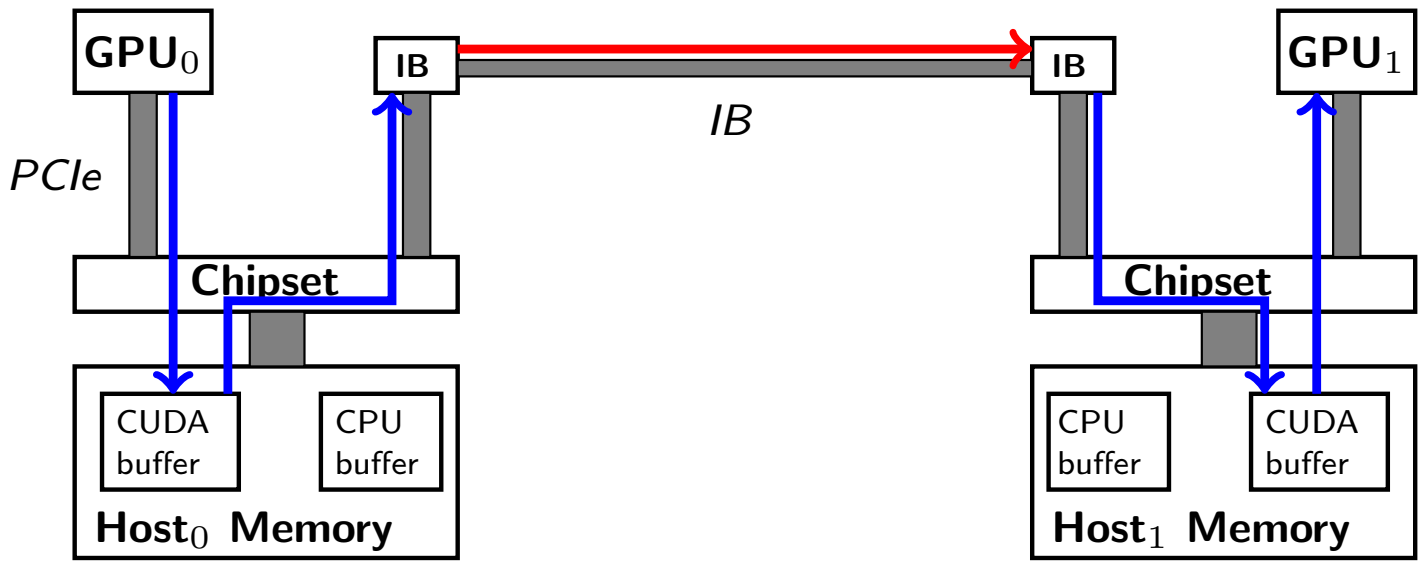
1. perform transfer in 3 steps:
   - a data copy from device to host using CUDA routines,
   - use MPI to copy data between hosts,
   - and a CUDA copy from host to destination device.



Inter-node GPU Communications

2 options:

1. perform transfer in 3 steps:
   - a data copy from device to host using CUDA routines,
   - use MPI to copy data between hosts,
   - and a CUDA copy from host to destination device.
2. Cuda-aware MPI feature (MPI+CUDA): address GPU buffers directly in MPI routines, data transfers are fully pipelined



Inter-node GPU Communications

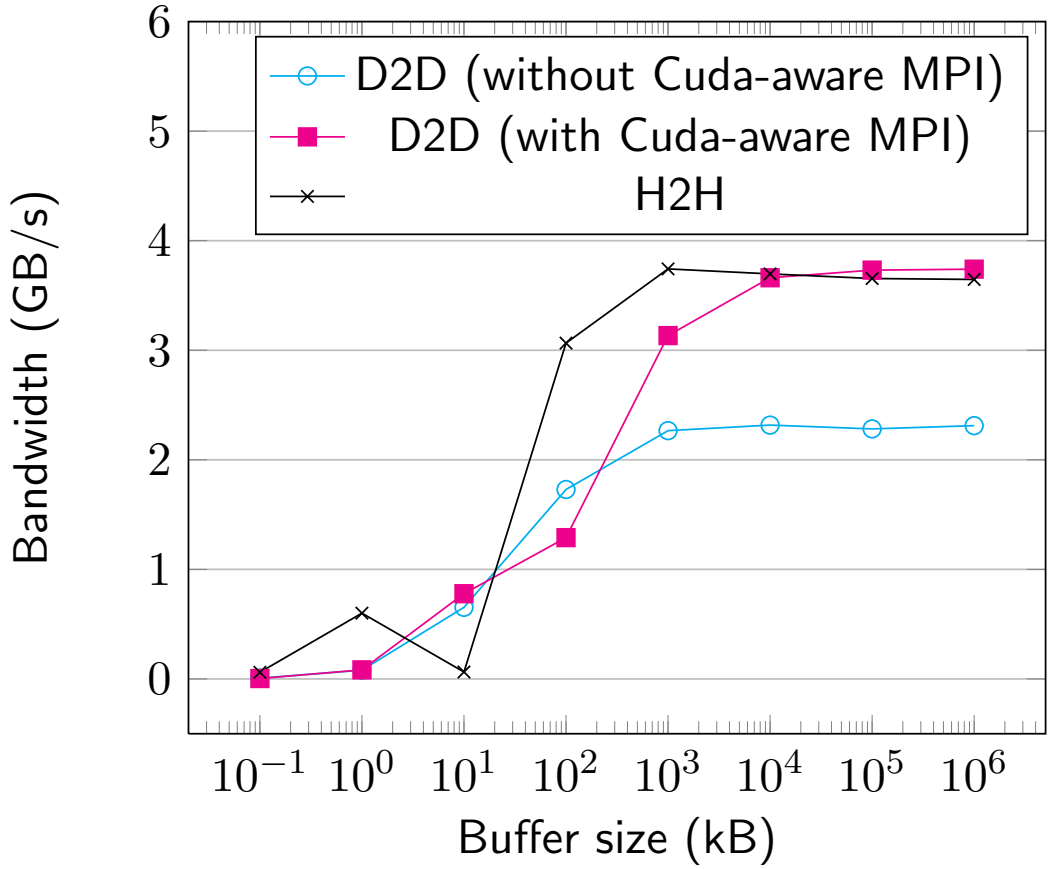


Table of Contents

DLP Record in GF$(2^{809})^\times$ [Caramel team 2013]

- Previous record was DLP in GF$(2^{613})^\times$ [Joux 2005].
- Matrix: 3.6M rows, with an average of 100 non-zero coefficients per row.
- Setup: 4-node cluster, connected with InfiniBand network at 40 Gb/s, each node equipped with 2 NVIDIA Tesla M2050 (3 GB).

Matrix size (Memory)	Possible blocking parameter	Overall comp. time	Ratio comm. /iteration
3.6M $\times$ 3.6M (3.2 GB)	$\boldsymbol{n = 4}$ **(2 GPUs/subtask)**	**4.5 days**	**16%**
	$n = 2$ (4 GPUs/subtask)	6 days	37%

DLP Record in GF$(2^{809})^\times$ [Caramel team 2013]

- Previous record was DLP in GF$(2^{613})^\times$ [Joux 2005].
- Matrix: 3.6M rows, with an average of 100 non-zero coefficients per row.
- Setup: 4-node cluster, connected with InfiniBand network at 40 Gb/s, each node equipped with 2 NVIDIA Tesla M2050 (3 GB).

Matrix size (Memory)	Possible blocking parameter	Overall comp. time	Ratio comm. /iteration
3.6M × 3.6M (3.2 GB)	$n = 4$ **(2 GPUs/subtask)**	**4.5 days**	**16%**
	$n = 2$ (4 GPUs/subtask)	6 days	37%
3M × 3M (2.7 GB)	$n = 8$ **(1 GPU/subtask)**	**2.5 days**	**0%**
	$n = 4$ (2 GPUs/subtask)	3 days	17%
	$n = 2$ (4 GPUs/subtask)	4.1 days	38%

DLP Record in a 596-bit Prime Field [Bouvier, Gaudry, Imbert, J., Thomé 2014]

- Previous record was 530 bit [Kleinjung 2007].
- Matrix: 7.3M rows with average weight of 150 coeff. per row (required mem. 9.8 GB).
- GPU Setup: 4-node cluster, connected with QDR Infiniband, each node equipped with 2 NVIDIA GeForce GTX 680 (4 GB).
- CPU Setup: 768-core cluster: 48 nodes connected with FDR Infiniband, each node hosts two 2-GHZ 8-core Intel Xeon E5-2650.

Setup	Approximate cost	Blocking parameter	Overall comp. time	Ratio com. /iteration
8 GPUs on 4 nodes	20k$	$n = 2$ (4 GPUs/subtask)	65 days	32%
768 cores on 48 nodes	**240k$**	$\boldsymbol{n = 12}$ **(64 cores/subtask)**	**39 days**	**19%**

DLP Record in a 596-bit Prime Field [Bouvier, Gaudry, Imbert, J., Thomé 2014]

- Previous record was 530 bit [Kleinjung 2007].
- Matrix: 7.3M rows with average weight of 150 coeff. per row (required mem. 9.8 GB).
- GPU Setup: 4-node cluster, connected with QDR Infiniband, each node equipped with 2 NVIDIA GeForce GTX 680 (4 GB).
- CPU Setup: 768-core cluster: 48 nodes connected with FDR Infiniband, each node hosts two 2-GHZ 8-core Intel Xeon E5-2650.

Setup	Approximate cost	Blocking parameter	Overall comp. time	Ratio com. /iteration
8 GPUs on 4 nodes	20k$	$n = 2$ (4 GPUs/subtask)	65 days	32%
768 cores on 48 nodes	**240k$**	$\boldsymbol{n = 12}$ **(64 cores/subtask)**	**39 days**	**19%**
96 GPUs on 48 nodes	+100k$	$n = 24$ (4 GPUs/subtask)	5.5 days	32%

Conclusion

- Parallelization of the matrix-vector product:
  - balance workloads over the computing nodes,
  - communication/computation scheme,
  - and several ways for intra- and inter-node communications.
- Optimal strategies to run this computation in cluster of GPUs or CPUs.

$\Rightarrow$ GPUs are the cost-efficient setup for the resolution.

Unfortunately,

2 Nvidia GeForce GTX 680 (Gamer's card) burned out.