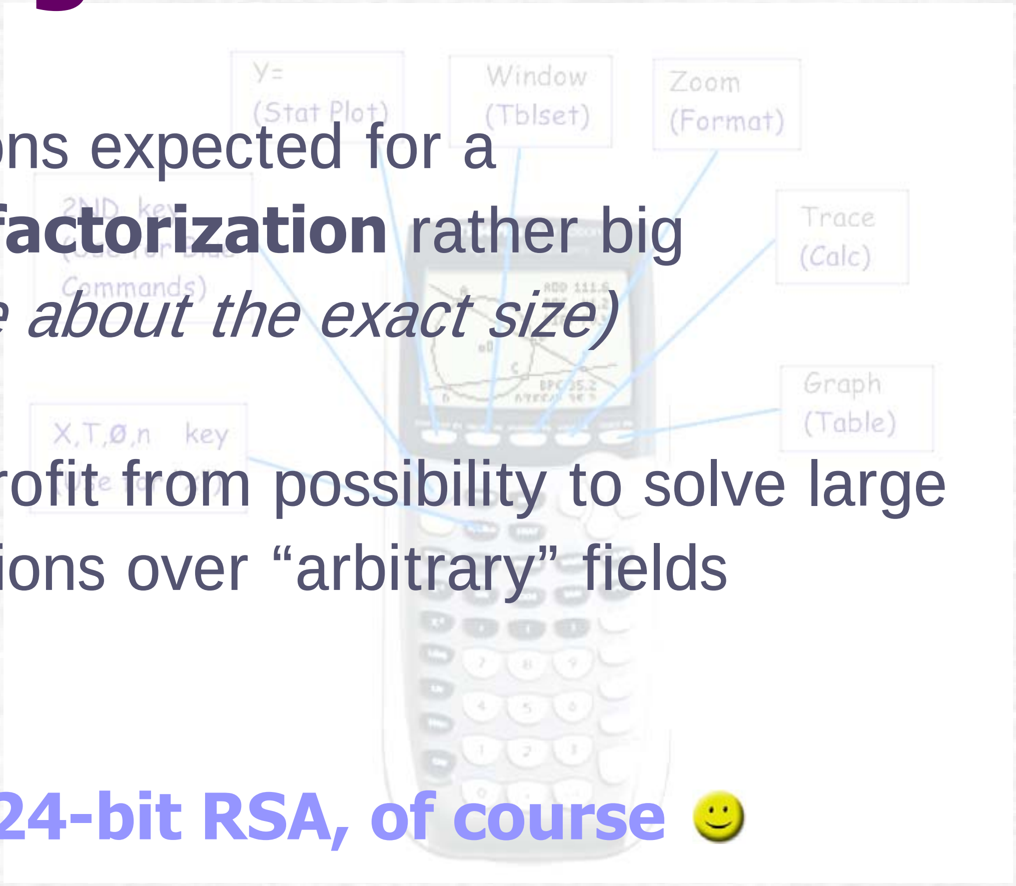
Dedicated Hardware to Solve Sparse Systems of Linear Equations: State of the Art & Application to Integer Factoring

Rainer Steinwandt

Florida Atlantic University, USA

(based on joint work with Willi Geiselmann, Adi Shamir, Eran Tromer)

Why linear algebra hardware?



- linear system of equations expected for a **1024 bit NFS-based factorization** rather big *(though one may argue about the exact size)*

- other algorithms may profit from possibility to solve large systems of linear equations over “arbitrary” fields (→[Frey04])

... key motivation is 1024-bit RSA, of course ☺

LA hardware: basic approach

Motivated by factoring with NFS, focus of LA hardware is on

(Block) Wiedemann algorithm for GF(2):

reduces NFS' LA step to iterated matrix-vector multiplications

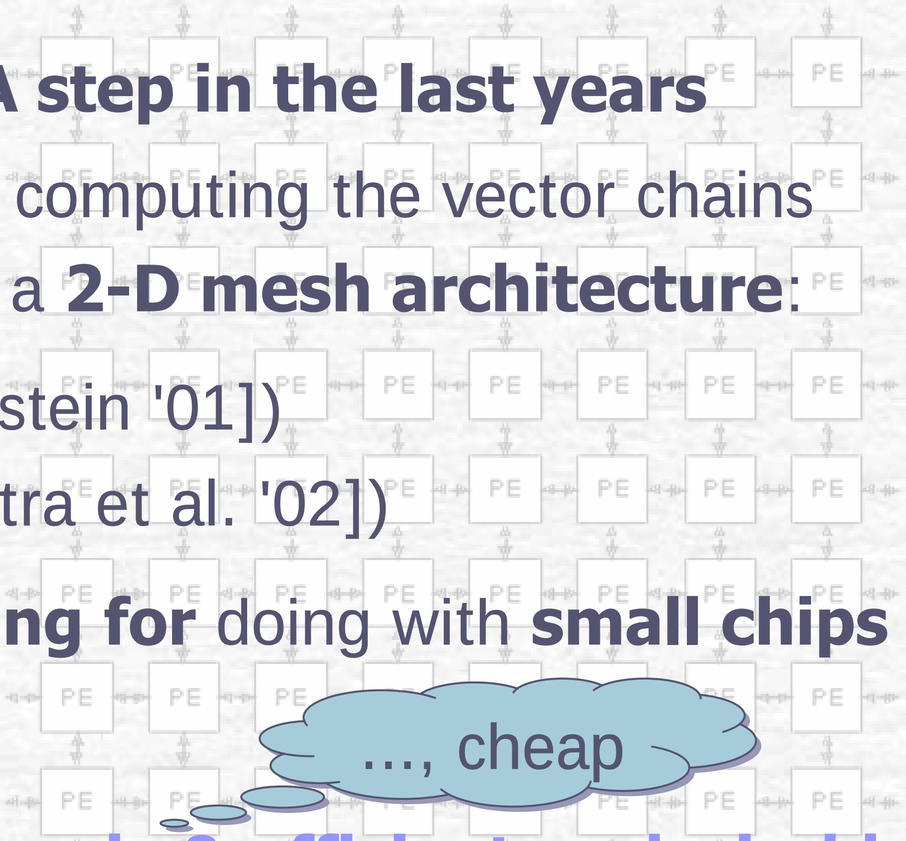
$$Av,\ A^2v,\ A^3v,\ \ldots,\ A^kv$$

with **sparse** (... **but potentially large**) matrix A

1024 bit: $A \in \mathrm{GF}(2)^{10^{10} \times 10^{10}}$

... but most recent design applies to other fields, too

LA & 2-D mesh architectures



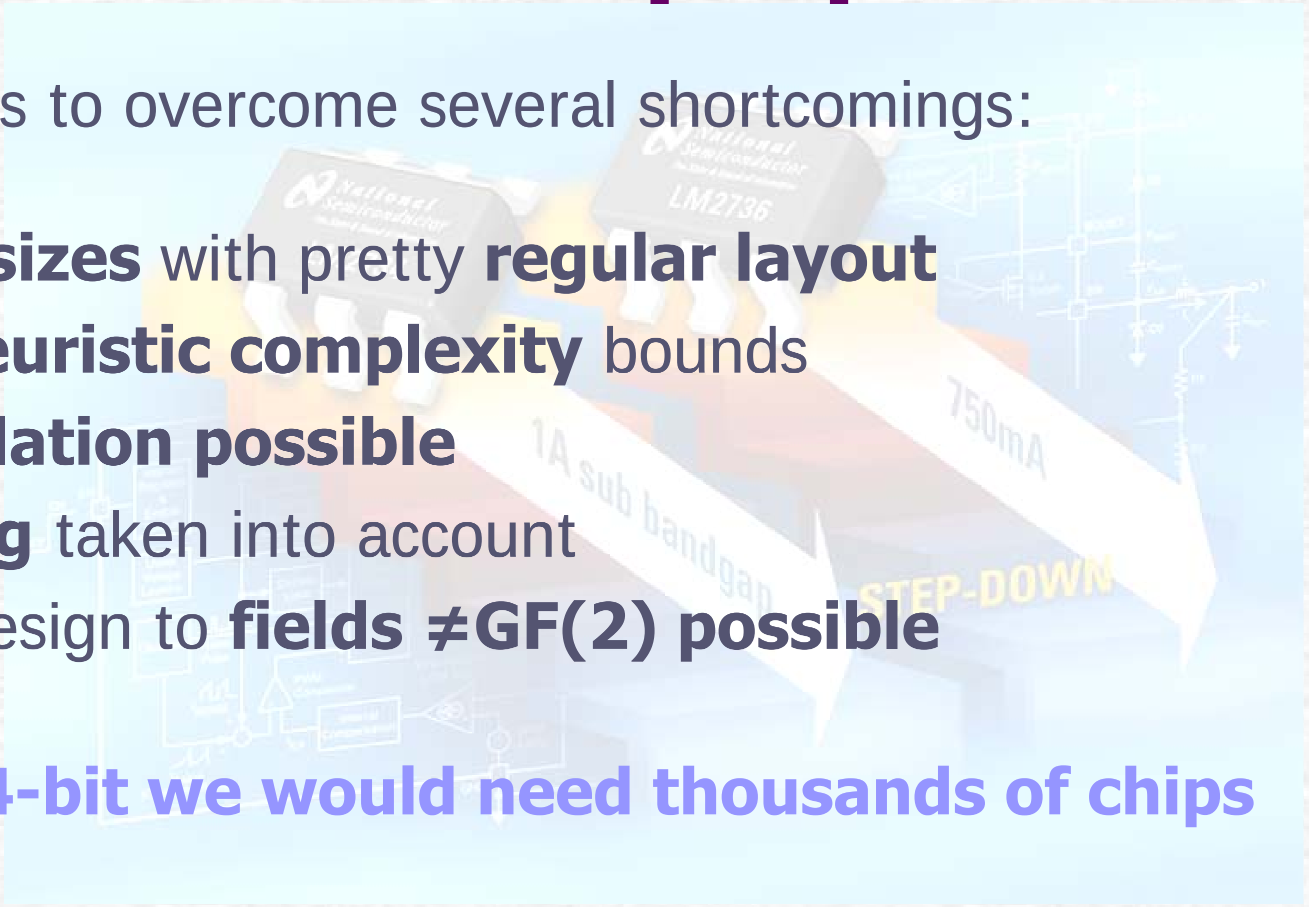
Devices proposed for the LA step in the last years

- offer methods for efficiently computing the vector chains $Av, A^2v, A^3v, \ldots, A^kv$ using a **2-D mesh architecture**:
  - 2-D **sorting** ($\rightarrow$[Bernstein '01])
  - 2-D **routing** ($\rightarrow$[Lenstra et al. '02])
- impose **another 2-D splitting for** doing with **small chips** ($\rightarrow$[Geiselmann, S. '03])

..., cheap

... not utopian, but not as simple & efficient as desirable

CHES '05: Another proposal

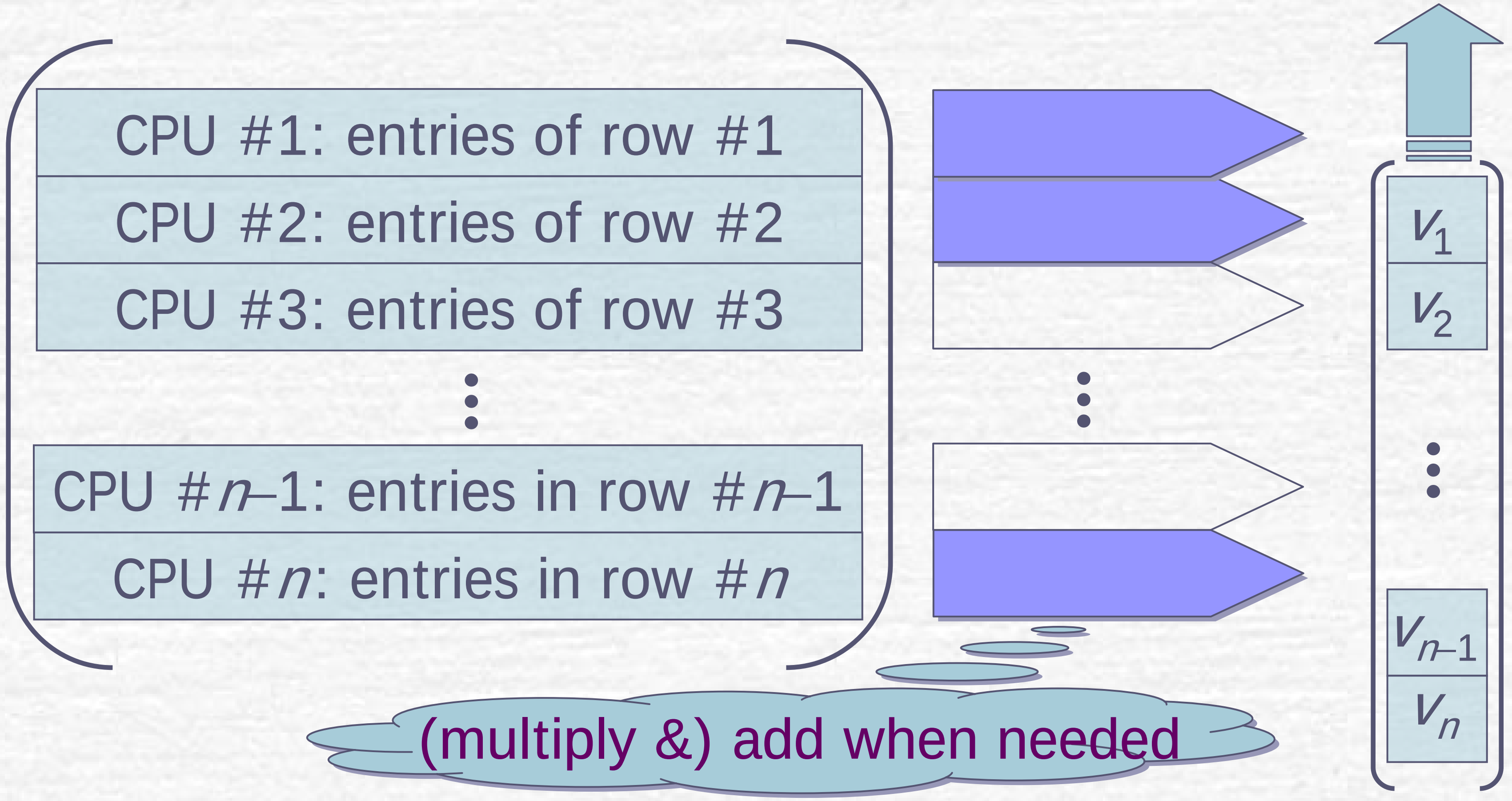


New design seems to overcome several shortcomings:

- **modest chip sizes** with pretty **regular layout**
- **no** need for **heuristic complexity** bounds
- software **simulation possible**
- **error handling** taken into account
- adapting the design to **fields ≠GF(2) possible**

… still, for 1024-bit we would need thousands of chips

Multiplying with $v \in \mathrm{GF}(q)^n$

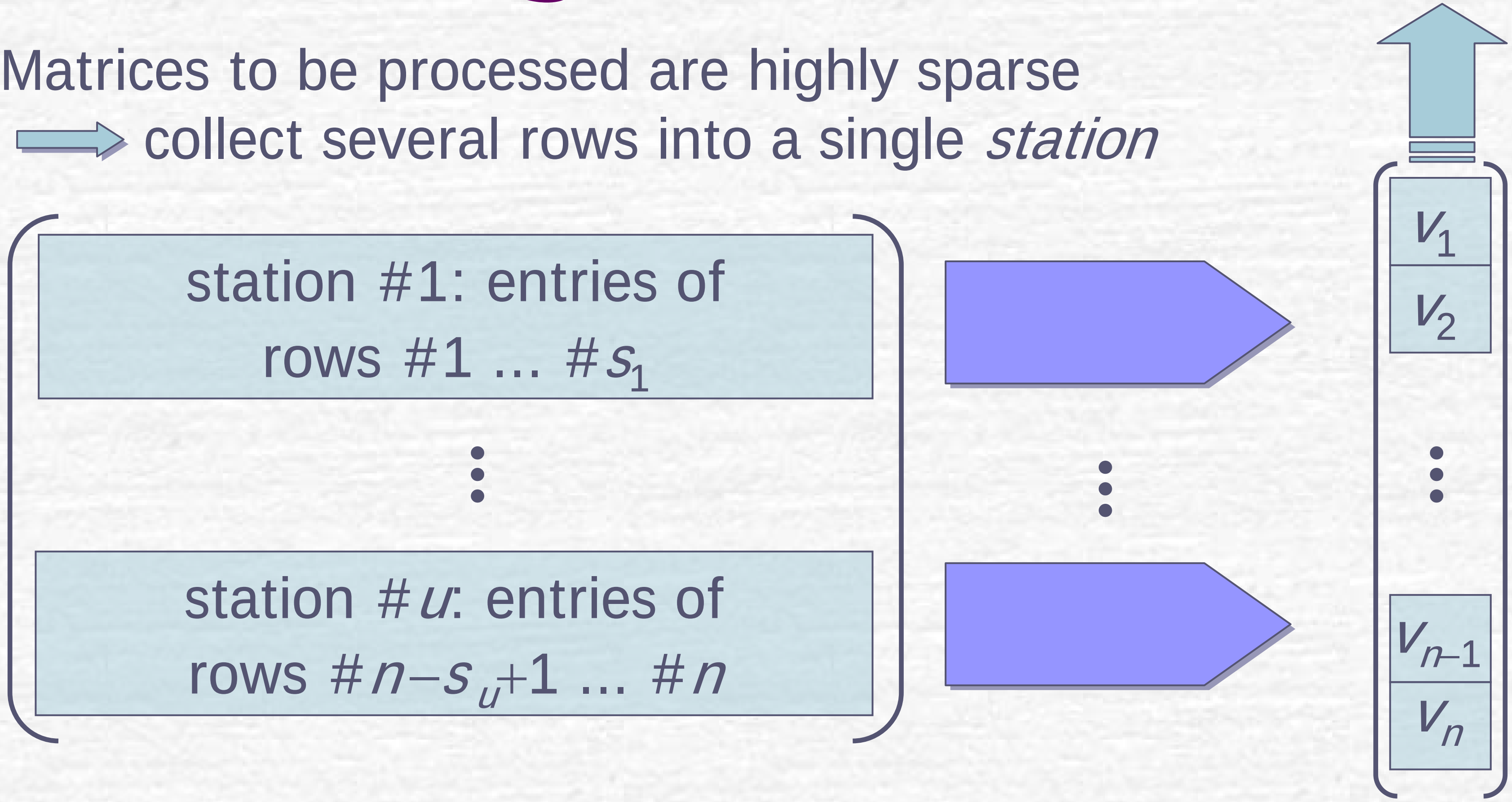


CPU #1: entries of row #1
CPU #2: entries of row #2
CPU #3: entries of row #3
⋮
CPU #n–1: entries in row #n–1
CPU #n: entries in row #n
v_1
v_2
⋮
v_{n-1}
v_n
(multiply &) add when needed

Collecting rows in stations

Matrices to be processed are highly sparse

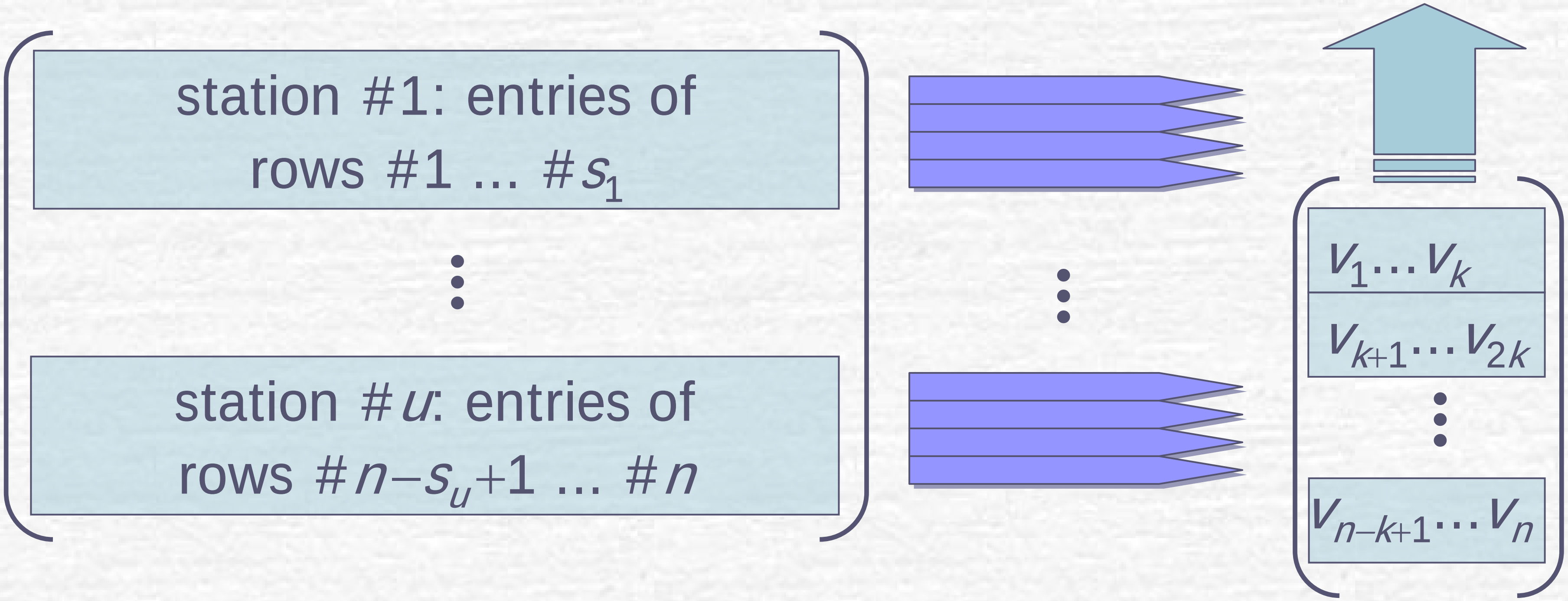
→ collect several rows into a single *station*


station #1: entries of rows #1 ... #s_1
⋮
station #u: entries of rows #$n-s_u+1$ ... #n
⋮
v_1
v_2
⋮
v_{n-1}
v_n

Additional parallelization

Needed arithmetics is not space-consuming

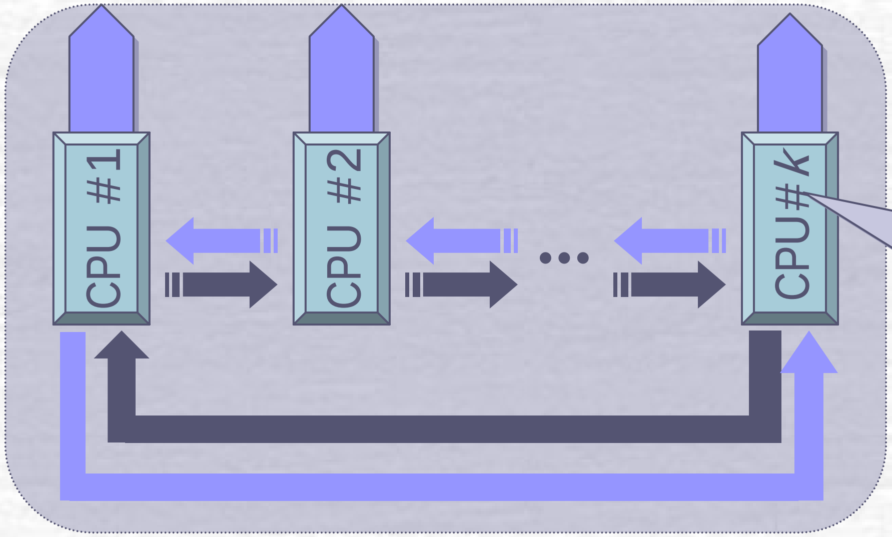
→ process $k>1$ vector components in parallel



station #1: entries of rows #1 ... #s_1
station #u: entries of rows #$n-s_u+1$ ... #n
$v_1 \ldots v_k$
$v_{k+1} \ldots v_{2k}$
$v_{n-k+1} \ldots v_n$

… using intra-station buses

Handling *k* vector components in parallel in each station:



Each CPU:

- s_i / k matrix rows
- GF(q)-multiplier (& -adder)

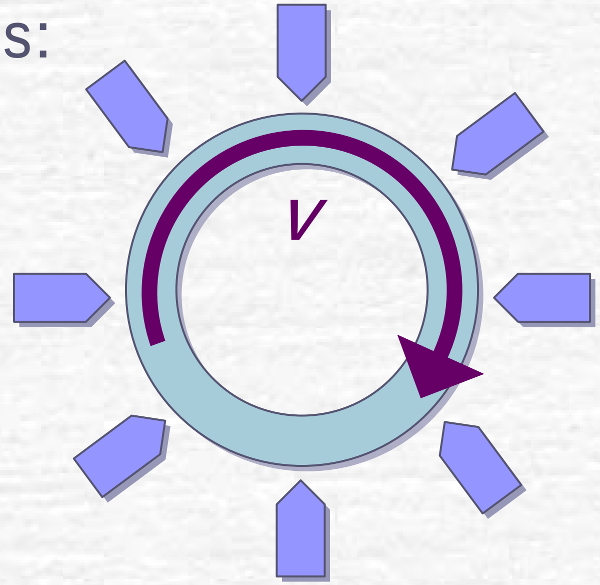
circular buses for intra-station transport of *v*-entries

Multiplying with *A* again

Actually needed: $A \cdot v$, $A \cdot Av$, $A \cdot A^2 v$, ...

⇨ result of multiplication must go back into vector pipeline

⇨ rearrange stations:



... have each station scan *v* in a different cyclic order

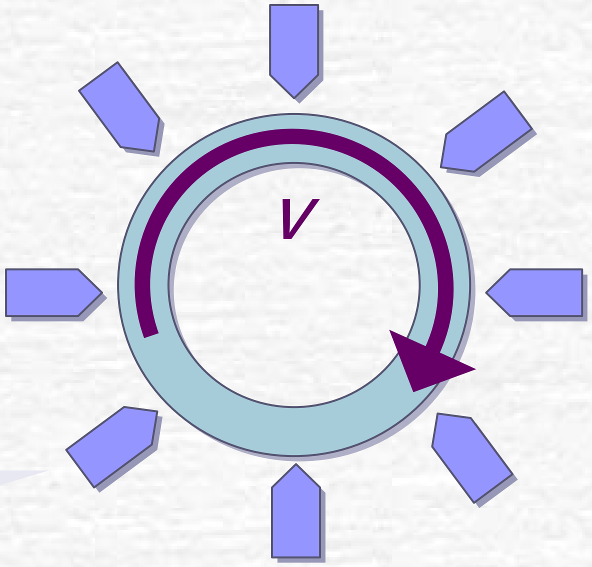
Doing another multiplication

GF(p)-addition commutative

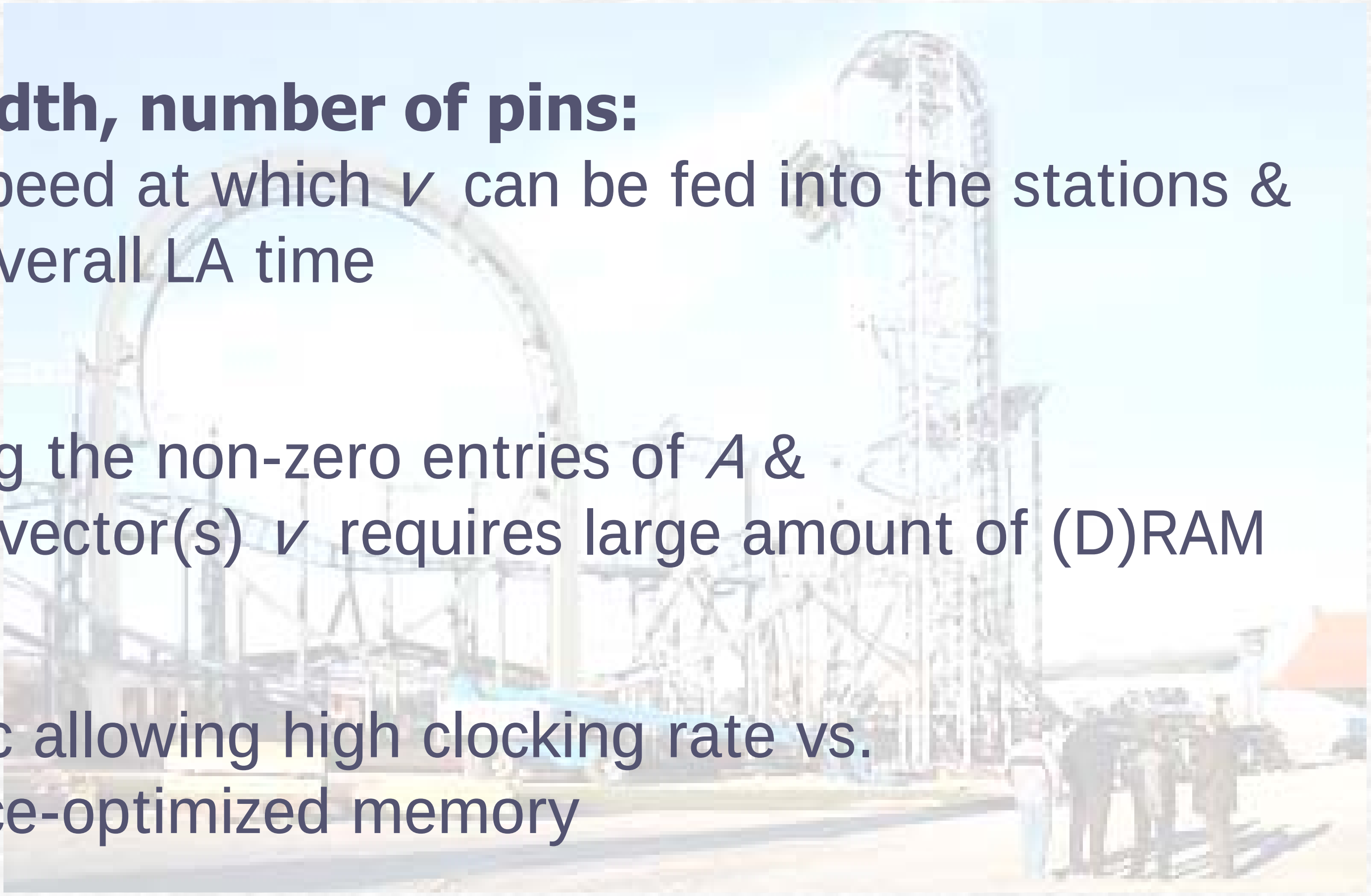
1 complete cycle yields $A \cdot v$

stations switch to 2nd mem. bank holding $A \cdot v$

Device is immmediately prepared for next multiplication.



Critical parameters



- **I/O Bandwidth, number of pins:**
  limits the speed at which v can be fed into the stations & therewith overall LA time
- **Memory:**
  representing the non-zero entries of A &
  storing the vector(s) v requires large amount of (D)RAM
- **Clock rate:**
  simple logic allowing high clocking rate vs.
  (slow) space-optimized memory

Techn(olog)ical limitations

- #pins limited through chip size (>2^{12} pins means large chips)
- logic for systolic design simpler than for mesh-based designs

$\Rightarrow$ increasing clocking rate to 1 GHz seems doable

What about the memory?

vector v: dense, 2×(D)RAM for n (=10^{10}) GF(q)-entries

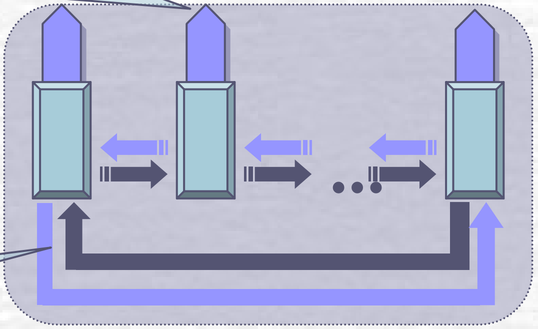
matrix A: GF(q)$^{\times}$-entry, row coord. within CPU, auxiliary flags

no need for random access, DRAM sufficient

Matrix handling

"External table" for reading *v*-entries:

#wait cycles	"read it" flag	bus no. to write on



"Internal table" for storing the matrix:

#wait cycles	"read it" flag	bus no. to read from
GF(q)$^\times$-entry	row coord.	"delete it" flag

Distributing the matrix

As with mesh based designs, we can **split *A* into submatrices** (→[Geiselmann, S. '03]):

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,r} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,r} \\ & \vdots & & \\ A_{s,1} & A_{s,2} & \cdots & A_{s,r} \end{pmatrix}, \quad v = \begin{pmatrix} v_{1,1} \\ \vdots \\ v_{1,r} \end{pmatrix} \cdots = \begin{pmatrix} v_{s,1} \\ \vdots \\ v_{s,r} \end{pmatrix}, \quad A \cdot v = \begin{pmatrix} \sum A_{1,j} \cdot v_{1,j} \\ \vdots \\ \sum A_{s,j} \cdot v_{s,j} \end{pmatrix}$$

store submatrix coordinates only

Block matrix multiplication

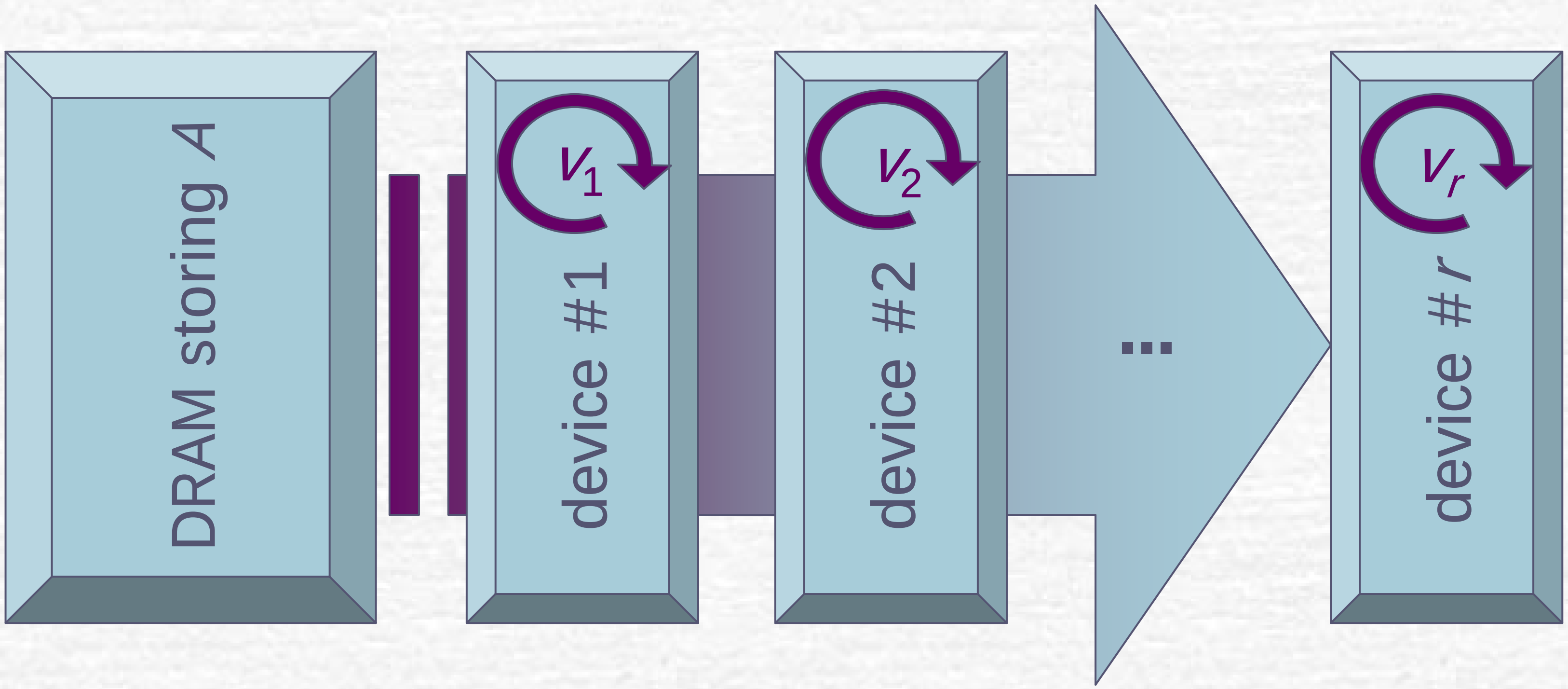
- assign a **multiplication circuit** to **each submatrix** $A_{i,j}$
- distribute/**load** appropriate ***v*-parts** into each circuit
- compute **all $A_{i,j} \cdot v_{i,j}$–values**
- **output** all **subproducts & add them** in a pipeline

result must be split & loaded into the device

Limiting factor for run time: I/O bandwidth/#pins

Systolic parallelization

Increased blocking factor without repeatedly storing *A*:



Combining it all?

splitting of *A* into submatrices can be **combined with systolic parallelization**

↓

short vectors + small matrices + simple logic

↓

small interconnected chips

↓

... may be fast, but not that trivial to implement

practical point of view: 2D-systolic looks preferable

1024-bit: what seems doable?

- Current manufacturing technology (90 nm, 1GHz, 1 cm^2,...):
  - **300x90 array of ASIC chips** (blocking factor *K*=900),
  - **each** (90-chip) **row fed by a 108-Gbit DRAM**,

  $\Rightarrow$ multiplication chains can be completed in ≈**2.4 months**

- Mesh-based design (90 nm, 200 Mhz, 85×85, **12.25 cm^2**,...):
  ≈**11.7 months**; throughput/silicon area worse by factor 6.5

… CHES ’05 design seems to be faster & more practical

What about errors?

- Uniform design offers **local fault tolerance:** on a faulty chip one can "bypass" faulty stations

- **High-level error recovery** remains **crucial:** running time of months is likely to involve errors

little extra hardware computing vector inner products allows reliable error detection $\longrightarrow$ "backtrack" to good state

Conclusion

- systolic design looks **preferable to mesh**-based approach: seems to be simpler, faster and require smaller chips
- topic of **"optimal" parameter choice** (purely systolic, matrix splitting, ...) deserves further exploration
- **small GF(2)-prototype** seems doable and desirable

... for factoring, improvements in sieving would be nice