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

\[ Av, A^2 v, A^3 v, \ldots, A^k v \]

with **sparse (… but potentially large)** matrix \( A \)

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

... but most recent design applies to other fields, too
Devices proposed for the LA step in the last years offer methods for efficiently computing the vector chains $A^v, A^2v, A^3v, \ldots, A^kv$ using a 2-D mesh architecture.

- 2-D sorting (→[Bernstein '01])
- 2-D routing (→[Lenstra et al. '02])

Impose another 2-D splitting for doing with small chips (→[Geiselmann, S. '03])

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

..., cheap
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 $v \in GF(q)^n$

- CPU #1: entries of row #1
- CPU #2: entries of row #2
- CPU #3: entries of row #3
- ... (multiply & add when needed)
- CPU #n−1: entries in row #n−1
- CPU #n: entries in row #n

\[
\begin{align*}
& v_1 \\
& v_2 \\
& \vdots \\
& v_{n-1} \\
& v_n
\end{align*}
\]
Collecting rows in stations

Matrices to be processed are highly sparse

collect several rows into a single station

station #1: entries of rows #1 \ldots \#s_1

\ldots

station #u: entries of rows #n−s_u+1 \ldots \#n
Additional parallelization

Needed arithmetics is not space-consuming

\[ \text{(process} \ k>1 \ \text{vector components in parallel)} \]

- Station #1: entries of rows #1, ..., #s₁
- Station #u: entries of rows #n−s₁+1, ..., #n

\[ \begin{align*}
V_1 & \ldots V_k \\
V_{k+1} & \ldots V_{2k} \\
& \vdots \\
V_{n-k+1} & \ldots V_n
\end{align*} \]
... using intra-station buses

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

Each CPU:
- $s_i/k$ matrix rows
- $\text{GF}(q)$-multiplier ($\&$ -adder)

Circular buses for intra-station transport of $\nu$-entries
Multiplying with $A$ again

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

- result of multiplication must go back into vector pipeline
- rearrange stations:

... have each station scan $v$ in a different cyclic order
Doing another multiplication

\[ \text{GF}(\mathcal{P}) \text{-addition commutative} \]

1 complete cycle yields \( A \cdot v \)

stations switch to 2\textsuperscript{nd} mem. bank holding \( A \cdot v \)

Device is immediately 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
- increasing clocking rate to 1 GHz seems doable

What about the memory?

**vector** $\mathbf{v}$: dense, $2 \times (D)$RAM for $n (=10^{10})$ $\text{GF}(q)$-entries

**matrix** $\mathbf{A}$: $\text{GF}(q)^{\times}$-entry, row coord. within CPU, auxiliary flags
no need for random access, DRAM sufficient
Matrix handling

"External table" for reading υ-entries:

<table>
<thead>
<tr>
<th>#wait cycles</th>
<th>&quot;read it&quot; flag</th>
<th>bus no. to write on</th>
</tr>
</thead>
</table>

"Internal table" for storing the matrix:

<table>
<thead>
<tr>
<th>#wait cycles</th>
<th>&quot;read it&quot; flag</th>
<th>bus no. to read from</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(q)×-entry</td>
<td>row coord.</td>
<td>&quot;delete it&quot; flag</td>
</tr>
</tbody>
</table>
Distributing the matrix

As with mesh based designs, we can split $A$ into submatrices (→[Geiselmann, S. '03]):
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 cm²,...):

- 300x90 array of ASIC chips (blocking factor $K=900$),
- each (90-chip) row fed by a 108-Gbit DRAM,
- multiplication chains can be completed in $\approx 2.4$ months

Mesh-based design (90 nm, 200 Mhz, 85×85, 12.25 cm²,...):
- $\approx 11.7$ 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 ➔ “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.