FFPACK: Finite Field Linear Algebra Package

Jean-Guillaume Dumas, Pascal Giorgi and Clément Pernet

pascal.giorgi@ens-lyon.fr, {Jean.Guillaume.Dumas, Clément.Pernet}@imag.fr
Motivation: Integer linear algebra
Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

⇒ Otherwise: Dense methods
Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

  ⇒ Otherwise: Dense methods

- Limit the growth of intermediate results:

  ⇒ Several computations over distinct finite fields and reconstruction using Chinese Remaindering
Introduction

Motivation: Integer linear algebra

- Sparse or structured matrix: specific methods (Blackbox,...)

⇒ Otherwise: Dense methods

- Limit the growth of intermediate results:

⇒ Several computations over distinct finite fields and reconstruction using Chinese Remaindering

Applications: integer polynomial factorization, Gröbner basis computation, integer system solving, ...
Exact Dense Linear Algebra Routines

**FFLAS**  Finite Field Linear Algebra Subroutines
- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm
**Exact Dense Linear Algebra Routines**

**FFLAS**  Finite Field Linear Algebra Subroutines
- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm

Matrix-matrix classic multiplication on a PII 735 MHz

- ATLAS (1)
- GF(19) --- Delayed with 32 bits (2)
- GF(19) --- Prime field on top of Atlas (3)
- GF(32) --- Galois field on top of Atlas (4)
Exact Dense Linear Algebra Routines

**FFLAS**  Finite Field Linear Algebra Subroutines
- Based on a Matrix Multiplication kernel
- Using numerical BLAS through conversions
- Fast Matrix Multiplication algorithm

**FFPACK**  Finite Field Linear Algebra Package
- Higher Level (cf LAPACK)
- Based on matrix triangularization

Matrix-matrix classic multiplication on a PIII 735 MHz
Contents

1. Base field representations
2. Triangular System Solve
   (a) Three implementations
   (b) Two cascade algorithms and comparison
3. Triangularizations
   (a) Three implementations and comparison
   (b) Dealing with data locality
4. Conclusions and Perspectives
Base field representation

- Modular<double>:
  - Based on machine double floating point representation
  - Only using the mantissa
    - $\Rightarrow$ Exact representation of integer up to $2^{53}$
  - Avoids conversions and extra memory storage when using FFLAS
Base field representation

- **Modular<double>:**
  - Based on machine `double` floating point representation
  - Only using the mantissa
    ⇒ Exact representation of integer up to $2^{53}$
  - Avoids conversions and extra memory storage when using FFLAS

- **Givaro-ZpZ:**
  - Based on machine integer (16, 32 or 64 bits)
  - Specialized dot-product (using delayed modulus)
Contents

1. Base field representations
2. Triangular System Solve
   (a) Three implementations
   (b) Two cascade algorithms and comparison
3. Triangularizations
   (a) Three implementations and comparison
   (b) Dealing with data locality
4. Conclusions and Perspectives
Triangular System Solve: \texttt{trsm}

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$. 
Triangular System Solve: \texttt{trsm}

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
Triangular System Solve: \texttt{trsm}

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms
Triangular System Solve: \texttt{trsm}

Compute a matrix $X \in \mathbb{K}^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms
- Three different approaches for exact computation over a finite field:
  1. A block recursive algorithm
  2. A wrapping of the BLAS \texttt{dtrsm}
  3. A matrix-vector based routine
Triangular System Solve: \texttt{trsm}

Compute a matrix $X \in K^{m \times n}$, s.t. $AX = B$.

- Used for numerical computations (in the BLAS)
- Building block for triangularization block algorithms
- Three different approaches for exact computation over a finite field:
  1. A block recursive algorithm
  2. A wrapping of the BLAS \texttt{dtrsm}
  3. A matrix-vector based routine
- Cascade algorithms as solution
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix}
= 
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]
1. The block recursive algorithm

\[
\begin{bmatrix}
  A_1 & A_2 \\
  0 & A_3
\end{bmatrix}
\begin{bmatrix}
  X_1 \\
  X_2
\end{bmatrix}
= 
\begin{bmatrix}
  B_1 \\
  B_2
\end{bmatrix}
\]

\(X_2 := \text{recursive call on } (A_3, B_2)\).
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix} = 
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

\(X_2 := \text{recursive call on } (A_3, B_2).\)
\(B_1 := B_1 - A_2X_2.\)
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix}
= \begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

\(X_2 :=\) recursive call on \((A_3, B_2)\).
\(B_1 := B_1 - A_2 X_2.\)
\(X_1 :=\) recursive call on \((A_1, B_1)\).
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix} =
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

\(X_2 := \text{recursive call on } (A_3, B_2)\).
\(B_1 := B_1 - A_2X_2\).
\(X_1 := \text{recursive call on } (A_1, B_1)\).

⇒ Reduces to matrix multiplication
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix} =
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

\(X_2 := \) recursive call on \((A_3, B_2)\).
\(B_1 := B_1 - A_2 X_2\).
\(X_1 := \) recursive call on \((A_1, B_1)\).

\(\Rightarrow\) Reduces to matrix multiplication
\(\rightarrow O(n^\omega)\) algebraic time complexity
1. The block recursive algorithm

\[
\begin{bmatrix}
A_1 & A_2 \\
0 & A_3
\end{bmatrix}
\begin{bmatrix}
X_1 \\
X_2
\end{bmatrix}
= 
\begin{bmatrix}
B_1 \\
B_2
\end{bmatrix}
\]

\[X_2 := \text{recursive call on } (A_3, B_2).\]
\[B_1 := B_1 - A_2X_2.\]
\[X_1 := \text{recursive call on } (A_1, B_1).\]

⇒ Reduces to matrix multiplication
→ \(O(n^\omega)\) algebraic time complexity
→ Efficiency of FFLAS
2. Wrapping the BLAS $\text{dtrsm}$

Same approach as for the matrix multiplication in FFLAS:

- Conversion: Finite Field $\rightarrow$ Real ($\text{double}$)
- Computation over the real (using BLAS $\text{dtrsm}$)
- Conversion: Real ($\text{double}$) $\rightarrow$ Finite Field
2. Wrapping the BLAS $\text{dtrsm}$

- Same approach as for the matrix multiplication in FFLAS:
  - Conversion: Finite Field $\rightarrow$ Real ($\text{double}$)
  - Computation over the real (using BLAS $\text{dtrsm}$)
  - Conversion: Real ($\text{double}$) $\rightarrow$ Finite Field

- Two constraints:
  - No division must occur during BLAS computation
  - No overflow
2. Wrapping the BLAS \texttt{dtrsm}

**First constraint:** Divisions must be exact in

\[
x_i = \frac{1}{a_{i,i}} \left( b_i - \sum_{j=i+1}^{n} a_{i,j} x_j \right)
\]

\( \Rightarrow A \) must have a unit diagonal.

\( \Rightarrow \) Precondition \( A \):

\[
\begin{align*}
U &= AD_A^{-1} \\
\text{solve } UY &= B \\
X &= D_A^{-1}Y
\end{align*}
\]

where \( D_A \) is the diagonal of \( A \).
2. Wrapping the BLAS \texttt{dtrsm}

Second constraint: No overflow during the BLAS computation

⇒ Must bound the growth of the coefficients:
2. Wrapping the BLAS \texttt{dtrsm}

Second constraint: No overflow during the BLAS computation

$\Rightarrow$ Must bound the growth of the coefficients:

- Naive: $(p - 1)p^{n-1} < 2^m$
Second constraint: No overflow during the BLAS computation

⇒ Must bound the growth of the coefficients:

- **Naive:** \((p - 1)p^{n-1} < 2^m\)

- Using a classical prime field repr.: \(0 ≤ x ≤ p - 1\):
  \[
  ⇒ \frac{p-1}{2} \left[ p^{n-1} + (p - 2)^{n-1} \right] < 2^m
  \]
2. Wrapping the BLAS \texttt{dtrsm}

Second constraint: No overflow during the BLAS computation

⇒ Must bound the growth of the coefficients:

- Naive: \((p - 1)p^{n-1} < 2^m\)

- Using a classical prime field repr.: \(0 \leq x \leq p - 1:\)
  \[
  \Rightarrow \frac{p-1}{2} \left[ p^{n-1} + (p - 2)^{n-1} \right] < 2^m
  \]

- Using a centered prime field repr.: \(-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}:\)
  \[
  \Rightarrow \frac{p-1}{2} \left( \frac{p+1}{2} \right)^{n-1} < 2^m
  \]
2. Wrapping the BLAS \texttt{dtrsm}

\Rightarrow \text{Limit the matrix order for a given prime } p:
2. Wrapping the BLAS \texttt{dtrsm}

\[ \Rightarrow \text{Limit the matrix order for a given prime } p: \]

For \( m = 53 \):

- \( p = 2 \Rightarrow n \leq 55 \)
- \( p = 9739 \Rightarrow n \leq 4 \)
- \( p = 94906249 \Rightarrow n \leq 2 \)
2. Wrapping the BLAS \texttt{dtrsm}

\[ \Rightarrow \text{Limit the matrix order for a given prime } p: \]

For \( m = 53: \)

- \( p = 2 \Rightarrow n \leq 55 \)
- \( p = 9739 \Rightarrow n \leq 4 \)
- \( p = 94906249 \Rightarrow n \leq 2 \)
3. Using Matrix-vector products

\[ X_i = B_i - A \cdot X_{i+1..n} \]

⇒ Matrix vector product

Different implementations:
3. Using Matrix-vector products

\[ \begin{align*}
X_i &= B_i - A \cdot X_{i+1..n} \\
\Rightarrow & \text{Matrix vector product}
\end{align*} \]

- Different implementations:
  - Using `modular<double>`: BLAS `gemv` and modulo
3. Using Matrix-vector products

\[ X_i = B_i - A.X_{i+1..n} \]

\[ \Rightarrow \text{Matrix vector product} \]

Different implementations:

- Using \texttt{modular<double>}: BLAS \texttt{gemv} and modulo
- Using integral representations: design of specialized dot-product routines [Dumas CASC’04]
3. Using Matrix-vector products

\[ X_i = B_i - A \cdot X_{i+1..n} \]

Matrix vector product

Different implementations:

- **Using modular<double>:** BLAS gemv and modulo
- **Using integral representations:** design of specialized dot-product routines [Dumas CASC’04]

Advantages:

- Delayed modulus (1 for each row of \( X \))
3. Using Matrix-vector products

\[ X_i = B_i - A.X_{i+1..n} \]

Matrix vector product

Different implementations:
- Using \texttt{modular<double>}: BLAS \texttt{gemv} and modulo
- Using integral representations: design of specialized dot-product routines [Dumas CASC’04]

Advantages:
- Delayed modulus (1 for each row of \(X\))
- nicer bound: \(n(p - 1)^2 < 2^m\)
3. Using Matrix-vector products

\[ X_i = B_i - A \cdot X_{i+1..n} \]

\( \Rightarrow \) Matrix vector product

Different implementations:
- **Using** \texttt{modular<double>: BLAS \emph{gemv} and modulo}
- **Using** integral representations: design of specialized dot-product routines [Dumas CASC’04]

Advantages:
- Delayed modulus (1 for each row of \( X \))
- nicer bound: \( n(p - 1)^2 < 2^m \)

Drawback: less efficient for large matrices \((n \geq 100)\)
Two cascade algorithms

Idea:

\[
\begin{align*}
&\text{trsm-rec} \quad \text{n:=n/2} \quad (n>n_{\text{blas}})\? \\
&\text{trsm-rec} \quad \text{n:=n/2} \quad (n>n_{\text{delayed}})\? \\
&\text{trsm-blas} \\
&\text{trsm-delayed}
\end{align*}
\]
Two cascade algorithms

Idea:

\[
\begin{align*}
& \text{trsm-rec} \\
& n := n/2 \\
& (n > n_{\text{blas}}) ? \\
& \text{YES} \\
& \text{NO} \\
& \text{trsm-blas}
\end{align*}
\]

Or

\[
\begin{align*}
& \text{trsm-rec} \\
& n := n/2 \\
& (n > n_{\text{delayed}}) ? \\
& \text{YES} \\
& \text{NO} \\
& \text{trsm-delayed}
\end{align*}
\]

Timings of the cascade algorithm over \(\mathbb{Z}/5\mathbb{Z}\) using \texttt{modular<double>} on a P4-2.4Ghz

- \texttt{trsm-rec}
- \texttt{trsm-blas}
- \texttt{trsm-delayed}

\[p = 5\]

Timings of the cascade algorithm over \(\mathbb{Z}/32749\mathbb{Z}\) using \texttt{Givaro-ZpZ} on a P4-2.4Ghz

- \texttt{trsm-rec}
- \texttt{trsm-blas}
- \texttt{trsm-delayed}

\[p = 5\]
Two cascade algorithms

Idea:

Or

Timings of the cascade algorithm over $\mathbb{Z}/5\mathbb{Z}$ using modular<double> on a P4−2.4GHz

Timings of the cascade algorithm over $\mathbb{Z}/32749\mathbb{Z}$ using Givaro−ZpZ on a P4−2.4GHz

modular<double> $p = 32749$

Givaro-ZpZ $p = 32749$
Conclusion

1. trsm-blas highly depends on the prime
   trsm-delayed do not
Conclusion

1. \texttt{trsm-blas} highly depends on the prime
   \texttt{trsm-delayed} do not

2. If the field representation can be chosen
   \Rightarrow \textbf{Use}\ Modular<\texttt{double}> \textbf{with} \texttt{trsm-blas}
Conclusion

1. $\text{trsm-blas}$ highly depends on the prime
   $\text{trsm-delayed}$ do not

2. If the field representation can be chosen
   $\Rightarrow$ Use $\text{Modular<double>}$ with $\text{trsm-blas}$

3. Otherwise
   $\Rightarrow$ For some cases, a specialization of dot-product can
   slightly outperform $\text{trsm-blas}$
Contents

1. Base field representations
2. Triangular System Solve
   (a) Three implementations
   (b) Two cascade algorithms and comparison
3. Triangularizations
   (a) Three implementations and comparison
   (b) Dealing with data locality
4. Conclusions and Perspectives
Triangularization

Specific issues:
- Have to deal with singular matrices
- Memory requirements
Triangularization

Specific issues:
- Have to deal with singular matrices
- Memory requirements

Provide a better analysis of the algebraic time complexity:
- improves the constant of the dominant term
- giving \( T = \frac{2}{3}n^3 + O(n^2) \) in the nonsingular case with classic matrix multiplication
Specific issues:
- Have to deal with singular matrices
- Memory requirements

Provide a better analysis of the algebraic time complexity:
- improves the constant of the dominant term
- giving $T = \frac{2}{3}n^3 + O(n^2)$ in the nonsingular case with classic matrix multiplication

We will compare 3 implementations:
- **LSP**: a block recursive algorithm [Ibara & Al.]
- **LUdivine**: LSP with lesser memory requirements
- **LQUP**: Fully in-place triangularization
LSP algorithms

LSP [Ibara]:

Split the row dimension

\[
\begin{bmatrix}
  & & A_1 & \\
  & S & & \\
  & & A_2 & \\
\end{bmatrix}
\]
LSP algorithms

LSP [Ibara]:

- Split the row dimension
- Recursive call on $A_1$
LSP algorithms

LSP [Ibara]:

- Split the row dimension
- Recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
LSP algorithms

LSP [Ibara]:

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
LSP algorithms

LSP [Ibara]:

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on $A_{22}$
LSP algorithms

LUdivine: result is in place

Split the row dimension

\[
\begin{bmatrix}
A1 \\
A2
\end{bmatrix}
\]
LSP algorithms

LUdvine: \textit{result is in place}

\begin{itemize}
  \item Split the row dimension
  \item recursive call on $A_1$
\end{itemize}
LSP algorithms

LUdivine: result is in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21} U_1^{-1}$
LSP algorithms

**LUdivine**: result is in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GW$
LSP algorithms

**LUdivine**: result is in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GW$
- recursive call on $A_{22}$
LSP algorithms

LQUP: *fully in place*

- Split the row dimension
LSP algorithms

LQUP: *fully in place*
- Split the row dimension
- Recursive call on $A_1$

```
L
|
U_1

V
```

```
L_1
|
|
```

```
A_{21}
|
A_{22}
```
LSP algorithms

**LQUP:** fully in place
- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21} U_1^{-1}$
**LSP algorithms**

**LQUP:** fully in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
LSP algorithms

LQUP: fully in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on $A_{22}$
LSP algorithms

LQUP: fully in place

- Split the row dimension
- recursive call on $A_1$
- $G \leftarrow A_{21}U_1^{-1}$
- $A_{22} \leftarrow A_{22} - GV$
- recursive call on $A_{22}$
- row permutations
## Comparisons

<table>
<thead>
<tr>
<th>n</th>
<th>1000</th>
<th>3000</th>
<th>5000</th>
<th>8000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSP</td>
<td>0.48</td>
<td>8.01</td>
<td>32.54</td>
<td>404.8</td>
<td>1804</td>
</tr>
<tr>
<td>LUdiveine</td>
<td>0.47</td>
<td>7.79</td>
<td>30.27</td>
<td>403.9</td>
<td>1691</td>
</tr>
<tr>
<td>LQUP</td>
<td>0.45</td>
<td>7.59</td>
<td>29.90</td>
<td>201.7</td>
<td>1090</td>
</tr>
</tbody>
</table>

- Similar timings when matrix fit in the RAM
- LQUP is slightly faster
- LQUP is fully in-place ⇒ no swap for $n = 8000$
Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

A triangularization algorithm: TURBO
Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

A triangularization algorithm: TURBO

P. Giorgi, J-G. Dumas & C. Pernet - FFPACK: Finite Field Linear Algebra Package – p.22/26
Dealing with data Locality

- Application: parallelism, out of core computations
- Use square recursive blocked data structure

A triangularization algorithm: TURBO

![Image of matrix order vs Mfops graph]

TURBO vs LQU for rank computation over $\mathbb{Z}_{101}$ on a P4-2.4Ghz, 512Mb RAM

P. Giorgi, J-G. Dumas & C. Pernet - FFPACK: Finite Field Linear Algebra Package – p.22/26
Conclusion

Results:

- Approach the timings of numerical routines:
  - 6.5s for a numeric LUP of a $3000 \times 3000$ matrix
  - 7.6s for a symbolic LQUP of a $3000 \times 3000$ matrix
Conclusion

Results:

- Approach the timings of numerical routines:
  - $6.5\, s$ for a numeric LUP of a $3000 \times 3000$ matrix
  - $7.6\, s$ for a symbolic LQUP of a $3000 \times 3000$ matrix
- Improved memory management of LSP factorization
- Further analysis of LSP time complexity.
Conclusion

Results:

- Approach the timings of numerical routines:
  - 6.5s for a numeric LUP of a $3000 \times 3000$ matrix
  - 7.6s for a symbolic LQUP of a $3000 \times 3000$ matrix

- Improved memory management of LSP factorization

- Further analysis of LSP time complexity.

- Optimal bounds for the coefficient growth in \texttt{trsm}
Conclusion

Results:

Approach the timings of numerical routines:
- 6.5s for a numeric LUP of a $3000 \times 3000$ matrix
- 7.6s for a symbolic LQUP of a $3000 \times 3000$ matrix

Improved memory management of LSP factorization

Further analysis of LSP time complexity.

Optimal bounds for the coefficient growth in \texttt{trsm}

Part of the LinBox library [\texttt{http://linalg.org}]
Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice.
- When not possible (ex LSP):
  - block recursive algorithms
Conclusion

Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice
- When not possible (ex LSP) ⇒ block recursive algorithms
- BLAS ⇒ No modulo
  ⇒ Need to control the coefficient growth
  ⇒ Bounds for correctness
Conclusion:

- Again: Wrapping numerical routines as much as possible appears to be the best choice
- When not possible (ex LSP) ⇒ block recursive algorithms
- BLAS ⇒ No modulo
  ⇒ Need to control the coefficient growth
  ⇒ Bounds for correctness
- Cascade structure
  ⇒ Switches between algorithms due to
    - Correctness constraints (theoretical thresholds)
    - Performance constraints (experimental thresholds)
Further developments

Self adapting software: automatic setup of optimal experimental thresholds
Further developments

- Self adapting software: automatic setup of optimal experimental thresholds
- Apply of the factorization to other applications: characteristic polynomial, null space, . . .
FFPACK: Finite Field Linear Algebra Package

Jean-Guillaume Dumas, Pascal Giorgi and Clément Pernet

pascal.giorgi@ens-lyon.fr, {Jean.Guillaume.Dumas, Clément.Pernet}@imag.fr