LinBox: a generic high performance library for exact linear algebra

Pascal Giorgi
Motivations

Exact linear algebra has become an important tool over the years e.g. cryptography, coding theory, experimental mathematics, etc. widely used in general computer algebra systems.
Exact linear algebra has become an important tool over the years, e.g., cryptography, coding theory, experimental mathematics, etc. It is widely used in general computer algebra systems. 

⇒ high performance implementations needed !!!
Motivations

Exact linear algebra has become a important tool over the years e.g. cryptography, coding theory, experimental mathematics, etc. widely used in general computer algebra systems.

⇒ high performance implementations needed !!!

<table>
<thead>
<tr>
<th>Numerical linear algebra</th>
<th>Exact linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ approximated solutions: float</td>
<td>⇒ exact solutions: ℤ, ℤₚ, ℚ, ℤ[ₓ]</td>
</tr>
<tr>
<td>✓ dedicated hardware</td>
<td>✓ no dedicated hardware</td>
</tr>
<tr>
<td>✗ pb of stability</td>
<td>✗ no stability issue</td>
</tr>
<tr>
<td>✓ mature developments</td>
<td>✓ slower development</td>
</tr>
</tbody>
</table>
Motivations

Exact linear algebra has become an important tool over the years, e.g., cryptography, coding theory, experimental mathematics, etc. widely used in general computer algebra systems.

⇒ high performance implementations needed !!!

<table>
<thead>
<tr>
<th>Numerical linear algebra</th>
<th>Exact linear algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇒ approximated solutions: float</td>
<td>⇒ exact solutions: ( \mathbb{Z}, \mathbb{Z}_p, \mathbb{Q}, \mathbb{Z}[X] )</td>
</tr>
<tr>
<td>✓ dedicated hardware</td>
<td>✗ no dedicated hardware</td>
</tr>
<tr>
<td>✗ pb of stability</td>
<td>✓ no stability issue</td>
</tr>
<tr>
<td>✓ mature developments</td>
<td>✓ improved over past 20 years</td>
</tr>
</tbody>
</table>

LinBox project has contributed a lot
Exact linear algebra versatility

\[
\begin{bmatrix}
993 & 512 & 509 \\
106 & 978 & 690 \\
946 & 442 & 832 \\
\end{bmatrix}^{-1} = \begin{bmatrix}
648 & 98 & 16 \\
648 & 839 & 305 \\
31 & 193 & 516 \\
\end{bmatrix} \quad \text{over } \mathbb{Z}_{997}
\]

\[
\begin{bmatrix}
\frac{14131}{9642515} & \frac{-11167}{19285030} & \frac{-8029}{19285030} \\
\frac{141137}{86782635} & \frac{172331}{173565270} & \frac{-157804}{86782635} \\
\frac{-219584}{86782635} & \frac{22723}{173565270} & \frac{458441}{173565270} \\
\end{bmatrix} \quad \text{over } \mathbb{Q}
\]

expression swell $\rightarrow$ op. on entries can be more than $O(1)$
Exact linear algebra versatility

Matrix storage → memory footprint can be $O(n)$

- algebraic vs bit (or word) complexity
- sparse vs dense vs structured matrix

} need different algorithms
Exact linear algebra versatility

Matrix storage $\rightarrow$ memory footprint can be $O(n)$

- algebraic vs bit (or word) complexity
- sparse vs dense vs structured matrix

Software challenge

A unified framework sustaining high performance
High performance linear algebra

exact computing ≠ numerical computing

- must tune arithmetic op. to benefit from hardware
- reductions to core problems ⇒ adaptative implem. with thresholds
High performance linear algebra

exact computing $\neq$ numerical computing

- must tune arithmetic op. to benefit from hardware
- reductions to core problems $\Rightarrow$ adaptative implem. with thresholds

### Major algorithmic reductions

- dense linear algebra in $O(n^\omega)$ with $\omega < 3$ [Strassen '69]
  - reduction to matrix mult. $\Rightarrow$ influence algebraic complexity
High performance linear algebra

exact computing ≠ numerical computing

- must tune arithmetic op. to benefit from hardware
- reductions to core problems ⇒ adaptative implem. with thresholds

Major algorithmic reductions

- dense linear algebra in $O(n^\omega)$ with $\omega < 3$ [Strassen '69]
  - reduction to matrix mult. ⇒ influence algebraic complexity

- sparse linear algebra in $O(\lambda n + n^2)$ [Wiedemann '86, Coppersmith '90]
  - reduction to SpMV/gcd ⇒ influence iterative methods for finite fields
High performance linear algebra

exact computing ≠ numerical computing

- must tune arithmetic op. to benefit from hardware
- reductions to core problems ⇒ adaptative implem. with thresholds

Major algorithmic reductions

- dense linear algebra in $O(n^\omega)$ with $\omega < 3$ [Strassen '69]
  
  reduction to matrix mult. ⇒ influence algebraic complexity

- sparse linear algebra in $O(\lambda n + n^2)$ [Wiedemann '86, Coppersmith '90]
  
  reduction to SpMV/gcd ⇒ influence iterative methods for finite fields

- dense lin.alg. with polynomials/integers in $O^\sim(n^{\omega d})$ [Storjohann '02]
  
  reduction to polynomials/integers matrix mult. ⇒ influence bit complexity
LinBox project

- Goes back to late ’90s !!!
  - founders: Giesbrecht, Kaltofen, Saunders, Villard
  - goal: a generic C++ library for blackbox linear algebra

\[ \mathbf{v} \in \mathbb{K}^n \quad \rightarrow \quad A \in \mathbb{K}^{m \times n} \quad \rightarrow \quad A\mathbf{v} \in \mathbb{K}^m \]

\[ \Rightarrow \text{mainly to exploit (block) Wiedemann’s approach} \]
LinBox project

- Goes back to late ’90s !!!
  - founders: Giesbrecht, Kaltofen, Saunders, Villard
  - goal: a generic C++ library for blackbox linear algebra

\[ \nu \in K^n \quad \rightarrow \quad A \in K^{m \times n} \quad \rightarrow \quad A\nu \in K^m \]

⇒ mainly to exploit (block) Wiedemann’s approach

- more than 20 years after:
  - main evolution: advocating new algorithms and high performance
  - an ecosystem of 3 open-source libraries: github.com/linbox-team
  - more than 40 contributors, but only few remain: Bouvier, Dumas, Giorgi, Pernet

⇒ acquired experience: algorithmic reductions are great in practice
Exact linear algebra reductions (in a nutshell)

**DENSE**
- HNF($\mathbb{Z}_p[X]$)
- SNF($\mathbb{Z}_p[X]$)
- Det($\mathbb{Z}_p[X]$)
- Linsys($\mathbb{Z}_p[X]$)
- O-Basis($\mathbb{Z}_p[X]$)
- HO-Lifting($\mathbb{Z}_p[[X]]$)
- MM($\mathbb{Z}_p[X]$)
- CRT($\mathbb{Z}$)

**SPARSE**
- Rank, Det, Charpoly, LinSys($\mathbb{Z}_p$)
- MinPoly($\mathbb{Z}_p$)
- SpMV($\mathbb{Z}_p$)

**modulo p polynomials**
- Inv($\mathbb{Z}_p[X]$)
- Nullspace($\mathbb{Z}_p[X]$)
- LinSys($\mathbb{Z}_p[X]$)
- MM($\mathbb{Z}_p[X]$)
- FFT($\mathbb{Z}_p[X]$)

**lintegers**
- Minpoly($\mathbb{Z}_p$)
- LinSys($\mathbb{Z}_p$)
- Det($\mathbb{Z}_p$)
- TRSM($\mathbb{Z}_p$)
- LU($\mathbb{Z}_p$)
- CharPoly($\mathbb{Z}_p$)
- HNF($\mathbb{Z}_p$)
- SNF($\mathbb{Z}_p$)

**HO-Lifting($\mathbb{Z}_p[[X]]$)**
- **O-Basis($\mathbb{Z}_p[X]$)**
- **Nullspace($\mathbb{Z}_p[X]$)**
- **Linsys($\mathbb{Z}_p[X]$)**
- **Det($\mathbb{Z}_p[X]$)**
- **HNF($\mathbb{Z}_p[X]$)**
- **SNF($\mathbb{Z}_p[X]$)**

6/36
Goal: make these reductions efficient in practice
⇒ "ease" software optimization process

Hierarchical development (mostly historical reason)
**Goal:** make these reductions efficient in practice
⇒ "ease" software optimization process

Hierarchical development (mostly historical reason)

- **Givaro**: basic arithmetic types/operations (e.g. rings)
**Goal:** make these reductions efficient in practice

⇒ "ease" software optimization process

Hierarchical development (mostly historical reason)

- **Givaro:** basic arithmetic types/operations (e.g. rings)
- **Fflas-ffpack:** dense linear algebra over finite fields
Goal: make these reductions efficient in practice
⇒ "ease" software optimization process

Hierarchical development (mostly historical reason)

- **Givaro**: basic arithmetic types/operations (e.g. rings)
- **Fflas-ffpack**: dense linear algebra over finite fields
- **LinBox**: linear algebra over general domains for dense/sparse/structured matrices
LinBox: a Middleware

- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

```
LinBox: a Middleware

- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

```

![Diagram of LinBox dependencies]

```c
#define MODULAR_DOUBLE

typedef Field Modular<double>

typedef Field F(17);

DenseMatrix<F,10,10> A(F,10,10);
DensePolynomial<F> Phi(F);
A.random();
charpoly(Phi, A);
```

```c
ffpack:

typedef Field Modular<double>

typedef Field F(17);
Poly1Dom<Field> R(F);
auto A = fflas(new(F,10,10)
RandomMatrix(F,10,10,A,10);
Poly1Dom<Field>::Element phi(11);
CharPoly(R, phi, 10,A,10);
```
LinBox: a Middleware

- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

```python
sage: A = matrix.random(GF(17), 10)
Phi = A.charpoly(algorithm="linbox")
```
LinBox: a Middleware

- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

---

**sage:**

```python
A = matrix.random(GF(17), 10)
Phi = A.charpoly(algorithm="linbox")
```

**linbox:**

```cpp
typedef Modular<double> Field;
Field F(17);
DenseMatrix<Field> A(F, 10, 10);
DensePolynomial<Field> Phi(F);
A.random();
charpoly(phi, A);
```
LinBox: a Middleware

- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

**sage:**
```
A = matrix.random(GF(17), 10)
Phi = A.charpoly(algorithm="linbox")
```

**linbox:**
```
typedef Modular<double> Field;
Field F(17);
DenseMatrix<Field> A(F, 10, 10);
DensePolynomial<Field> Phi(F);
A.random();
charpoly(phi, A);
```

**ffpack:**
```
typedef Modular<double> Field;
Modular<Field> F(17);
Poly1Dom<Field> R(F);
auto A = fflas_new(F, 10, 10);
RandomMatrix(F, 10, 10, A, 10);
Poly1Dom<Field>::Element phi(11);
CharPoly(R, phi, 10, A, 10);
```
Outline

Which genericity in LinBox and how?

How LinBox gets high-performance for dense linear algebra mod $p$?
Which genericity in LinBox and how?

How LinBox gets high-performance for dense linear algebra mod $p$?
Arithmetic is provided within a domain: \( D \cdot \text{add}(c, a, b) \)

- finite fields/rings: \( \mathbb{Z}/p\mathbb{Z}, \mathbb{Z}/m\mathbb{Z} \) (supporting multi-precision)
- extension fields: \( \text{GF}(q^k) \) (characteristic < 16-bits)
- integers, rationals (wrapping GMP library)

\( \hookrightarrow \) shipped with Givaro library

Standardized domain API: easy generic code through template

- encapsulation of element type as Element
- op. result as first parameter (pre C++11 std::move)
- ...

Goal \( \Rightarrow \) provide solid foundation for basic arithmetic
A central object in LinBox workflow (FFLAS-FFPACK → LinBox → SageMath) → API for field arithmetic $\mathbb{Z}/p\mathbb{Z}$
A central object in LinBox workflow (FFLAS-FFPACK → LinBox → SageMath) ↪ API for field arithmetic $\mathbb{Z}/p\mathbb{Z}$

defined as \texttt{Modular<Storage\_t, Compute\_t> F(p)};

- \texttt{Storage\_t}: type of field elements
- \texttt{Compute\_t}: type of interm. result, $xy + z \leq p(p - 1)$ no overflow
- the prime $p$ is only stored once in F
A central object in LinBox workflow (FFLAS-FFPACK → LinBox → SageMath) ↪ API for field arithmetic $\mathbb{Z}/p\mathbb{Z}$

defined as $\text{Modular<Storage}_t, \text{Compute}_t> F(p);$

- $\text{Storage}_t$: type of field elements
- $\text{Compute}_t$: type of interm. result, $xy + z \leq p(p-1)$ no overflow
- the prime $p$ is only stored once in $F$

**Example**

```cpp
Modular<
uint16_t,
uint32_t>
F(65521);         // 16-bits prime max
Modular<
uint16_t,
uint32_t> :: Element x, y, z;

F.init(x, 212121); F.init(y, 12);       // reduce x, y modulo 65521
F.axpyin(x, y, y);                    // x=x+y*y mod 65521
```
Wide coverage of native machine types:

Modular<\texttt{float}, \texttt{float}> // 12−bits prime max
Modular<\texttt{uint32\_t}, \texttt{uint32\_t}> // 16−bits prime max
Modular<\texttt{float}, \texttt{double}> // 24−bits prime max
Modular<\texttt{double}, \texttt{double}> // 26−bits prime max
Modular<\texttt{uint32\_t}, \texttt{uint64\_t}> // 32−bits prime max

⇒ ModularBalanced<\ldots> : centered encoding \([−\frac{p-1}{2}, \frac{p-1}{2}]\)
Wide coverage of native machine types:

- `Modular<float, float>` // 12−bits prime max
- `Modular<uint32_t, uint32_t>` // 16−bits prime max
- `Modular<float, double>` // 24−bits prime max
- `Modular<double, double>` // 26−bits prime max
- `Modular<uint32_t, uint64_t>` // 32−bits prime max

⇒ `ModularBalanced<...>`: centered encoding $[-\frac{p-1}{2}, \frac{p-1}{2}]$

Use C++11 `enable_if` and type traits:

- to restrict code bloat: `Compute_t` and `Storage_t` must be consistent
- to share generic implementation:

```cpp
std::enable_if<std::is_integral<_Storage_t>::value and
               std::is_integral<_Compute_t>::value and
               (sizeof(_Storage_t) == sizeof(_Compute_t) or
               2* sizeof(_Storage_t) == sizeof(_Compute_t)))>::type
```
Givaro: extending the precision

- using GMP multiprecision integers: `Integers`
- using own recursive fixed size integers: `ruint<K>`
  \[ \textcolor{orange}{ruint<K>} = \textcolor{orange}{ruint<K-1>} | \textcolor{orange}{ruint<K-1>} \]
- modular with Error Free transform for FP: `ModularExtended<double>`
  \[ a \times b = c + d \text{ where } c = a \otimes b \text{ and } d = FMA(a, b, -c) = a \otimes b \ominus c \]

```
Modular <ruint<7>,ruint<7>> // 2^6–bits prime max
Modular <ruint<7>,ruint<8>> // 2^7–bits prime max
Modular <Integers,Integers> // multiprecision
ModularExtended<double> // 53–bits prime max
```
Givaro: extending the precision

- using GMP multiprecision integers: \texttt{Integers}
- using own recursive fixed size integers: \texttt{ruint<K>}
  \[ \Rightarrow \texttt{ruint<K>} = \texttt{ruint<K-1>}|\texttt{ruint<K-1>}\]
- modular with Error Free transform for FP: \texttt{ModularExtended<double>}
  \[ \Rightarrow a \times b = c + d \text{ where } c = a \otimes b \text{ and } d = \text{FMA}(a, b, -c) = a \otimes b \ominus c \]

\begin{align*}
\text{Modular <ruint <7>,ruint <7> >} & \quad // \text{2^6—bits prime max} \\
\text{Modular <ruint <7>,ruint <8> >} & \quad // \text{2^7—bits prime max} \\
\text{Modular <Integers,Integers> } & \quad // \text{multiprecision} \\
\text{ModularExtended<double> } & \quad // \text{53—bits prime max} \\
\end{align*}

- Fixed size or multiprecision integers through: \texttt{ZRing<Compute_t>}
  \[ \rightarrow \text{ZRing<Integers> for } \mathbb{Z} \]
Exemple of generic code with Givaro

template <typename Domain>
void dotProduct(Domain::Element& res, 
               const Domain &D, 
               const std::vector<Domain::Element>& u, 
               const std::vector<Domain::Element>& v)
{
    D.init(res, D.zero);
    for (int i = 0; i < u.size(); i++)
        D.axpyin(res, u[i], v[i])
    return res;
}

using finite field

Modular<float> GF(17)
vector<float> u(10), v(10);
float d;
dotProduct(d, u, v);

using integers

ZRing<Integer> Z
vector<Integer> u(10), v(10);
Integers d;
dotProduct(d, u, v);
Which genericity in LinBox and how?

How LinBox gets high-performance for dense linear algebra mod $p$?
What is provided?

- high-performance matrix multiplication
- tuned reductions to matrix multiplication mod $p$
- Minpoly($\mathbb{Z}_p$)
- CRT($\mathbb{Z}_p$)
- LU($\mathbb{Z}_p$)
- Det($\mathbb{Z}_p$)
- TRSM($\mathbb{Z}_p$)
- LinSys($\mathbb{Z}_p$)
- CharPoly($\mathbb{Z}_p$)
- MM($\mathbb{Z}_p$)
- MM($\mathbb{Z}_p$)

Main ingredients:
- delegate some optimization to BLAS library:
  - ✓ cache re-use
  - subcubic matrix multiplication (Strassen-Winograd)
- generic interface for Intel SIMD intrinsic (SSE/AVX/AVX2/AVX512)
- PALADIn: PArallel Linear Algebra Dedicated Interface
What is provided?

- high-performance matrix multiplication
- tuned reductions to matrix mul: minimizing $\text{mod } p$/memory
FFLAS-FFPACK: dense linear algebra mod $p$

What is provided?

- high-performance matrix multiplication
- tuned reductions to matrix mul: minimizing mod $p$/memory

Main ingredients

- delegate some optimization to BLAS library: ✓ cache re-use
- subcubic matrix multiplication (Strassen-Winograd)
- generic interface for Intel SIMD intrinsic (SSE/AVX/AVX2/AVX512)
- PALADIn: PArallel Linear Algebra Dedicated Interface
Main operation: **AXPY**: $a \times b + c$
Main operation: **AXPY: \(a \times b + c\)**

- Machine word arithmetic for exact matrix multiplication

- Many BLAS libraries available: OpenBlas, BLIS, MKL, etc.
Main operation: **AXPY**: \( a \times b + c \)

FP numbers seems the good choice !!!

✓ Many BLAS libraries available: OpenBlas, BLIS, MKL, etc.
Machine word arithmetic for matrix multiplication $\mod p$

Modular reduction is delayed after few AXPYS: $\sum a_{i,k}b_{k,j} < 2^\beta$

$\Rightarrow$ limit $p$ to half wordsize

![Graphs showing speed and bit speed vs. Modulo p (bitsize)]

*benchmark on Intel Sandy Bridge (courtesy of C. Pernet)*

- best performances with FP (except in corner cases)
- double precision delivers highest bit op. throughput
Matrix multiplication \( \text{mod} p (< 26\text{bits}) \)

- delayed reductions \( \text{mod} \ p \) with SIMD optimisation \( \checkmark \ O(n^2) \)
- adaptative multiplication over \( \mathbb{Z} \)
  - \( t \) levels of Strassen-Winograd if \( 9^t \left\lfloor \frac{n}{2^t} \right\rfloor (p - 1)^2 < 2^{53} \) \( \checkmark \ \omega < 3 \)
  - use BLAS as base case \( \checkmark \ \text{cache+simd} \)

---

![Benchmark Graph](benchmark.png)

*benchmark on Intel Haswell, \( p < 20 \) bits*
**FFLAS-FFPACK: API design**

- template interface inspired from BLAS: explicit 1D array with strides, dims
  
  $A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$

- most optimizations use static type of $\mathbb{Z}_p$ (not the value of $p$)
  
  ⇒ type traits to specialized template functions: fgemm, ftrsm, etc.
FFLAS-FFPACK: API design

- template interface inspired from BLAS: explicit 1D array with strides, dims

\[
A = \begin{bmatrix}
a_{00} & a_{01} & a_{02} \\
a_{10} & a_{11} & a_{12} \\
\end{bmatrix}
\]

- most optimizations use static type of $\mathbb{Z}_p$ (not the value of $p$)

  $\Rightarrow$ type traits to specialized template functions: \texttt{fgemm}, \texttt{ftrsm}, etc.

---

Modular\texttt{<double>} F(65521); // prime field over double
auto A = fflas_new(F,10,20); // A is 10x20 matrix
auto B = fflas_new(F,20,30); // B is 20x30 matrix
auto C = fflas_new(F,10,30); // C is 10x30 matrix

// compute $C = A \times B = (0 \times C + 1 \times A \times B )$
fgemm(F,FflasNoTrans,FflasNoTrans,10,30,20,F.one,A,10,B,20,F.zero,C,30);

fflas_delete(A); fflas_delete(B); fflas_delete(C);
Use algorithmic reduction to \texttt{fgemm}
⇒ but minimize modular reductions
Dense linear algebra modulo $p$: implementation approach

Use algorithmic reduction to $\text{fgemm}$
$\Rightarrow$ but minimize modular reductions

Example with $\text{ftrsm}$:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

- $A_3X_2 = B_2 \mod p$
- $D = B_1 - A_2X_2 \text{ over } \mathbb{Z}$
- $A_1X_1 = D \mod p$

reduce r.h.s mod$p$ when $n$ small enough (solve over $\mathbb{Z}$)
Dense linear algebra modulo $p$: implementation approach

Use algorithmic reduction to $\text{fgemm}$
⇒ but minimize modular reductions

Example with $\text{ftrsm}$:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$$

- $A_3X_2 = B_2 \mod p$
- $D = B_1 - A_2X_2$ over $\mathbb{Z}$
- $A_1X_1 = D \mod p$

reduce r.h.s $\mod p$ when $n$ small enough (solve over $\mathbb{Z}$)

✓ only $O(n^2)$ modular reductions
✓ practical performance $\sim \text{fgemm}$
LQUP/PLUQ factorization reduces to \texttt{fgemm} and \texttt{ftrsm}.
Dense linear algebra modulo $p$: implementation approach

**LQUP/PLUQ factorization** reduces to \texttt{fgemm} and \texttt{ftrsm}

**Tile Iterative**

**Slab Recursive**

**Tile Recursive**

- getrf: $A \to L, U$
- trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$
- gemm: $C \leftarrow C - A \times B$

**careful choice to**
- minimize $\text{mod } p$ [Dumas, Pernet, Sultan 13]
- benefit more from Strassen/Winograd
Dense linear algebra modulo $p$ ($<26\text{ bits}$): reductions in practice

\textit{benchmark on Apple M1 Max laptop - 1 core (AMX - 2022), }$p = 131071$
Dense linear algebra modulo $p$: parallelism in practice

Benchmark on Intel SandyBridge - 32 core (AVX - 2015) courtesy of C. Pernet
Matrix multiplication \textit{mod} p (≥ 32\textit{bits})

No more native op. (e.g. \(\mathbb{Z}_{1267650600228229401496703205653}\))

\(\Rightarrow\) GMP library \(\Rightarrow\) costly: no SIMD, bad cache reuse

\(\Rightarrow\) Givaro::ruint<K> better but still costly: no SIMD
Matrix multiplication $\mod p \ (\geq 32 \text{ bits})$

No more native op. (e.g. $\mathbb{Z}_{1267650600228229401496703205653}$)
$\Rightarrow$ GMP library $\rightarrow$ costly: no SIMD, bad cache reuse
$\Rightarrow$ Givaro::ruint<K> better but still costly: no SIMD

Most efficient solutions $\Rightarrow$ reduction to smaller prime(s) matrix mult.

- convert to polynomial matrix mult. $\mod q$ (Kronecker)
  $\mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \mathbb{Z}_q[X]_{<d} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}_p$

- convert to many matrix multip. $\mod p_i$ (CRT)
  $\mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \mathbb{Z}_{p_1 \times \cdots \times p_d} \rightarrow \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_d} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}_p$

$AB \mod (p_1 \times \cdots \times p_d) \leftrightarrow (AB \mod p_1, \ldots, AB \mod p_d)$
Matrix multiplication $\text{mod } p \ (\geq 32\text{bits})$

No more native op. (e.g. $\mathbb{Z}_{126765060022829401496703205653}$)  
\[ \Rightarrow \text{GMP library} \rightarrow \text{costly: no SIMD, bad cache reuse} \]  
\[ \Rightarrow \text{Givaro::ruint<K> better but still costly: no SIMD} \]

Most efficient solutions  \[ \Rightarrow \text{reduction to smaller prime(s) matrix mult.} \]

- convert to polynomial matrix mult. $\text{mod } q$ (Kronecker)  
  \[ \mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \mathbb{Z}_q[X]_{<d} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}_p \]

- convert to many matrix multip. $\text{mod } p_i$ (CRT)  
  \[ \mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \mathbb{Z}_{p_1 \times \cdots \times p_d} \rightarrow \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_d} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}_p \]

RNS conversions  
\[ AB \mod (p_1 \times \cdots \times p_d) \leftrightarrow (AB \mod p_1, \ldots, AB \mod p_d) \]

How to improve the reduction? especially RNS
Fast RNS conversions $O(d \log(d) \log \log(d))$ word op. [Borodin, Moenck 74]

$\Rightarrow$ hard to optimize in practice
Optimizing RNS conversions

Fast RNS conversions $O(d \log(d) \log \log(d))$ word op. [Borodin, Moenck 74]  
$\Rightarrow$ hard to optimize in practice

Naive RNS conversions $O(d^2)$ word op.
Optimizing RNS conversions

Fast RNS conversions $O(d \log(d) \log \log(d))$ word op. [Borodin, Moenck 74]  
⇒ hard to optimize in practice

Naive RNS conversions $O(d^2)$ word op.  
⇒ can be reduced to matrix mult. for many conversion [DGLS18]

■ pseudo-reduction:

$$A_0 + A_1 \beta + \cdots A_{d-1} \beta^{d-1} \rightarrow \begin{bmatrix} A_0 & \ldots & A_{d-1} \end{bmatrix} \times \begin{bmatrix} \beta^i \mod p_j \end{bmatrix}_{i,j}$$

$O(d) \rightarrow O(\log d)$

■ $r$ RNS conversions: $O(rd^{\omega-1}) + O(d^2)$ word op.
Simultaneous conversions with RNS: in practice

One can extend the \( p_j \) without sacrificing too much performance

\[ \Rightarrow \text{doubling the prime size halves the number of moduli} \]
Simultaneous conversions with RNS: in practice

**Reduction to RNS**

**Reconstruction from RNS**

Benchmark on Apple M1 Max laptop - for matrix multiplication \( (n = 16) \)

One can extend the \( p_j \) without sacrificing too much performance

\[ \Rightarrow \text{doubling the prime size halves the number of moduli} \]
Main difficulties

- must fit the FFLAS API: to not re-implement algo. reductions
- offering cache efficiency
- allow to use word-size dgemm/fgemm without overhead
FFLAS: RNS implementation

Main difficulties

- must fit the FFLAS API: to not re-implement algo. reductions
- offering cache efficiency
- allow to use word-size dgemm/fgemm without overhead

Our solution:

- array of residues with stride
- two matrix linearizations
  ⇒ contiguous scalar/matrix residues
- redefinition of pointer/iterator
  ⇒ handling RNS strides: \( \text{ptr} + i, \ast \text{ptr} \)
- fix \( \beta = 2^{16} \) and 26-bits moduli
our solution: use multi-modular approach \( O(n^\omega d + n^2 d^{\omega-1}) \)
\[ \rightarrow \text{reduce everything to dgemm} \]
FFLAS integer matrix multiplication

our solution: use multi-modular approach \( O(n^{\omega}d + n^2d^{\omega-1}) \)

\( \rightarrow \) reduce everything to dgemm

benchmark on Apple M1 Max laptop - 1 core (AMX - 2022)
FFLAS integer matrix multiplication

our solution: use multi-modular approach $O(n^\omega d + n^2d^{\omega-1})$

$\hookrightarrow$ reduce everything to \texttt{dgemm}

benchmark on Apple M1 Max laptop - 1 core (AMX - 2022)

over $\mathbb{Z}_p$: reduce afterward (small slowdown)

$\Rightarrow$ could be slightly improved by incorporating mod$p$ during CRT
Dense linear algebra modulo $p$ (> 32 bits): on today laptop

Goes from Modular<Integers> to RnsInteger<rns_double> domain

$\hookrightarrow$ apply our generic reductions codes

---

**Reduction to Matrix Multiplication (mod 128 bits prime)**

**GFLOPS**

- **fgemm FLAS**
- **ftrsm FFLAS**
- **fpluq FFLAS**
- **fgemm FLINT**
- **ftrsm FLINT**
- **fpluq FLINT**

**Matrix Dimensions**

- 128
- 256
- 512
- 1024
- 2048
- 4096

**benchmark on Apple M1 Max laptop - 1 core (AMX - 2022)**

$\Rightarrow$ **Timings:** $1024 \times 1024$ matrices in less than a second
Some remarks

- The regime for primes between 32-bits and 64-bits not satisfactory
- Hybrid RNS (fast/gemm) could be beneficial for large integers
- Belief that double has better bitspeed than float is no more true: IA/ML sneaks into the game, and architecture follows the market

⇒ Apple M1 Max processor is 4× faster on float than double
Some remarks

- The regime for primes between 32-bits and 64-bits not satisfactory
- Hybrid RNS (fast/gemm) could be beneficial for large integers
- Belief that double has better bitspeed than float is no more true: IA/ML sneaks into the game, and architecture follows the market ⇒ Apple M1 Max processor is 4× faster on float than double
Thank you !!!
Simultaneous conversions with RNS: main idea

Let $||AB||_\infty < M = \prod_{i=1}^{d} m_i < \beta^d$ with coprime $m_i < \beta$.

Multi-reduction of a single entry

Let an integer $a = a_0 + a_1 \beta + \cdots + a_{d-1} \beta^{d-1}$ to reduce mod $m_i$ then

$$
\begin{bmatrix}
|a|_{m_1} \\
\vdots \\
|a|_{m_d}
\end{bmatrix} =
\begin{bmatrix}
1 & |\beta|_{m_1} & \cdots & |\beta^{d-1}|_{m_1} \\
\vdots & \vdots & \ddots & \vdots \\
1 & |\beta|_{m_d} & \cdots & |\beta^{d-1}|_{m_d}
\end{bmatrix}
\times
\begin{bmatrix}
a_0 \\
\vdots \\
ad_{d-1}
\end{bmatrix}
- \begin{bmatrix}
q_1 m_1 \\
\vdots \\
q_d m_d
\end{bmatrix}
$$

with $|q_i m_i| < d \beta^2$

**pseudo-reduction:** size $O(d) \Rightarrow$ size $O(\log d)$

**Lemma:** computing $A$ and $B$ modulo the $m_i$’s costs

$O(n^2 d^{\omega-1} + n^2 d M(\log d) + d^2)$ word op.
Simultaneous conversions with RNS: CRT

CRT formulae: \( a = \left( \sum_{i=1}^{k} |aM_{i}^{-1}|_{m_{1}} \cdot M_{i} \right) \mod M \) with \( M_{i} = M/m_{i} \)

Reconstruction of a single entry

Let \( M_{i} = \alpha_{0}^{(i)} + \alpha_{1}^{(i)} \beta + \cdots + \alpha_{d-1}^{(i)} \beta^{d-1} \), then

\[
\begin{bmatrix}
    a_{0} \\
    \vdots \\
    a_{d-1}
\end{bmatrix} =
\begin{bmatrix}
    \alpha_{0}^{(1)} & \cdots & \alpha_{0}^{(d)} \\
    \vdots & \ddots & \vdots \\
    \alpha_{d-1}^{(1)} & \cdots & \alpha_{d-1}^{(d)}
\end{bmatrix}
\times
\begin{bmatrix}
    |aM_{1}^{-1}|_{m_{1}} \\
    \vdots \\
    |aM_{d}^{-1}|_{m_{d}}
\end{bmatrix}
\]

with \( a_{i} < d\beta^{2} \) and \( a = a_{0} + \cdots + a_{k-1}\beta^{k-1} \mod M \).

Lemma: retrieving \( AB \) from its images modulo the \( m_{i} \)’s costs \( O(n^{2}d^{\omega-1} + n^{2}d \log d + d^{2}) \) word op.