LinBox: a generic high performance library for exact linear algebra

Pascal Giorgi

RTCA Lyon - 29 june 2023

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.

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.

$\Rightarrow$ 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.

$\Rightarrow$ high performance implementations needed !!!

Numerical linear algebra

$\Rightarrow$ approximated solutions : float

- ✓ dedicated hardware
- ✗ pb of stability
- ✓ mature developments

Exact linear algebra

$\Rightarrow$ exact solutions: $\mathbb{Z}, Z_p, \mathbb{Q}, \mathbb{Z}[X]$

- ✗ no dedicated hardware
- ✓ no stability issue
- ✗ slower development

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 !!!

Numerical linear algebra

⇒ approximated solutions : float

- ✓ dedicated hardware
- ✗ pb of stability
- ✓ mature developments

Exact linear algebra

⇒ exact solutions: $\mathbb{Z}, Z_p, \mathbb{Q}, \mathbb{Z}[X]$

- ✗ no dedicated hardware
- ✓ no stability issue
- ✓ improved over past 20 years

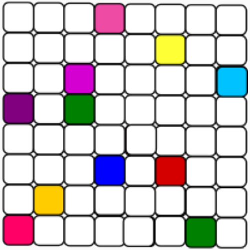
LinBox project has contributed a lot

Exact linear algebra versatility

$$\begin{bmatrix} 993 & 512 & 509 \\ 106 & 978 & 690 \\ 946 & 442 & 832 \end{bmatrix}^{-1} = \begin{cases} \begin{bmatrix} 648 & 98 & 16 \\ 648 & 839 & 305 \\ 31 & 193 & 516 \end{bmatrix} \text{ over } \mathbb{Z}_{997} \\ \\ \begin{bmatrix} \frac{14131}{9642515} & -\frac{11167}{19285030} & -\frac{8029}{19285030} \\ \frac{141137}{86782635} & \frac{172331}{173565270} & -\frac{157804}{86782635} \\ -\frac{219584}{86782635} & \frac{22723}{173565270} & \frac{458441}{173565270} \end{bmatrix} \text{ over } \mathbb{Q} \end{cases}$$

expression swell → op. on entries can be more than $O(1)$

Exact linear algebra versatility

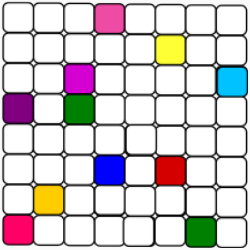


matrix storage → memory footprint can be $O(n)$

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

} need different algorithms

Exact linear algebra versatility



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

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

} need different algorithms

Software challenge

a unified framework sustaining high performance

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

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

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
- sparse linear algebra in $O(\lambda n + n^2)$ [Wiedemann '86, Coppersmith '90]
  reduction to SpMV/gcd $\Rightarrow$influence iterative methods for finite fields
- dense lin.alg. with polynomials/integers in $\tilde{O}(n^\omega d)$ [Storjohann '02]
  reduction to polynomials/integers matrix mult. $\Rightarrow$influence bit complexity

LinBox project

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

$\Rightarrow$mainly to exploit (block) Wiedemann's approach

LinBox project

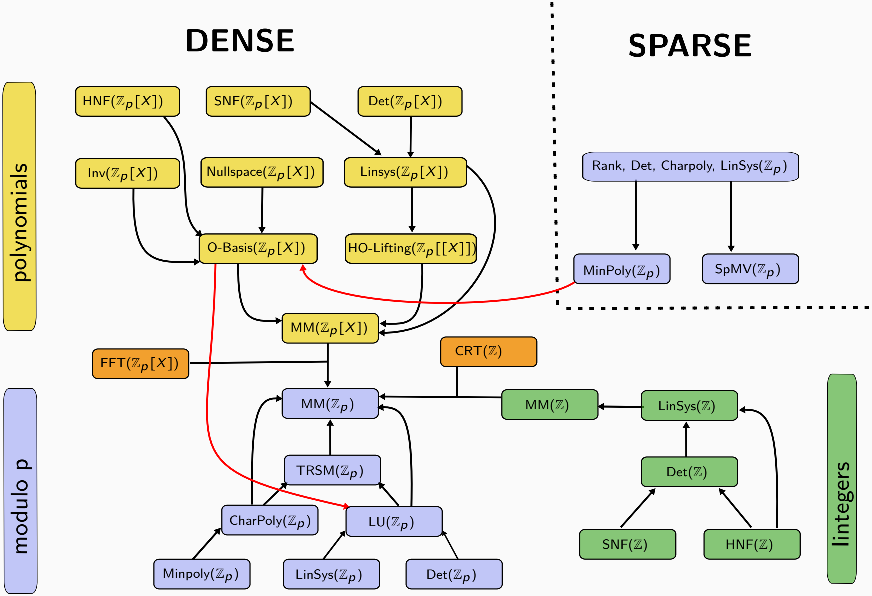
- Goes back to late '90s !!!
  - founders: Giesbrecht, Kaltofen, Saunders, Villard
  - goal: a generic C++ library for blackbox linear algebra

⇒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)



LinBox: an ecosystem of C++ libraries

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

Hierarchical development (mostly historical reason)

LinBox: an ecosystem of C++ libraries

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

Hierarchical development (mostly historical reason)

- Givaro: basic arithmetic types/operations (e.g. rings)

LinBox: an ecosystem of C++ libraries

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: an ecosystem of C++ libraries

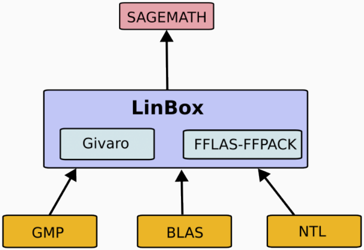
Goal: make these reductions efficient in practice
⇒ "ease" software optimization process

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

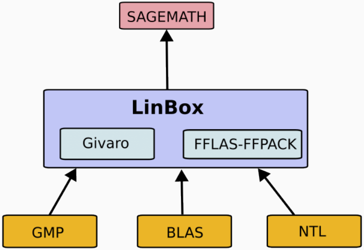
- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software



LinBox: a Middleware

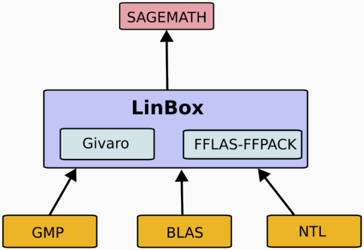
- C++ API ensure genericity through template code
- rely on some other libraries: to get functionalities/performance
- interface to general mathematical software

```
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

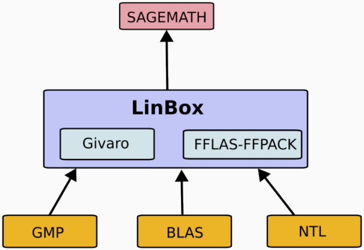


sage:
```
A=matrix.random(GF(17),10)
Phi=A.charpoly(algorithm="linbox")
```

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

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 ?

Outline

Which genericity in LinBox and how ?

How LinBox gets high-performance for dense linear algebra mod p ?

Exemple: basic arithmetic

Arithmetic is provided within a domain: `D.add(c,a,b)`

- finite fields/rings : $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{Z}/m\mathbb{Z}$ (supporting multi-precision)
- extension fields : $\mathrm{GF}(q^k)$ (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

Givaro: the `Modular <...>` class

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

Givaro: the `Modular <...>` class

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

defined as `Modular<Storage_t, Compute_t> F(p);`

- `Storage_t` : type of field elements
- `Compute_t` : type of interm. result, $xy + z \leq p(p-1)$ no overflow
- the prime p is only stored once in `F`

Givaro: the `Modular<...>` class

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

defined as `Modular<Storage_t, Compute_t> F(p);`

- `Storage_t` : type of field elements
- `Compute_t` : type of interm. result, $xy + z \leq p(p-1)$ no overflow
- the prime p is only stored once in `F`

Example

```
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
```

Givaro: the `Modular <...>` class

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-bitsprime max
Modular<uint32_t,uint64_t >    // 32-bits prime max
```

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

Givaro: the Modular <...> class

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-bitsprime 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:

```
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>`
  $\Rightarrow$ `ruint<K>` = `ruint<K-1>|ruint<K-1>`
- modular with Error Free transform for FP: `ModularExtended<double>`
  $\Rightarrow$ $a \times b = c + d$ where $c = a \otimes b$ 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: `Integers`
- using own recursive fixed size integers: `ruint<K>`
  $\Rightarrow$ `ruint<K>` = `ruint<K-1>|ruint<K-1>`
- modular with Error Free transform for FP: `ModularExtended<double>`
  $\Rightarrow$ $a \times b = c + d$ where $c = a \otimes b$ 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
```

- Fixed size or multiprecision integers through: `ZRing<Compute_t>`
  $\hookrightarrow$ `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<Integers> Z
vector<Integers> u(10),v(10);
Integers d;
dotProduct(d,u,v);
```

Outline

Which genericity in LinBox and how ?

How LinBox gets high-performance for dense linear algebra mod p ?

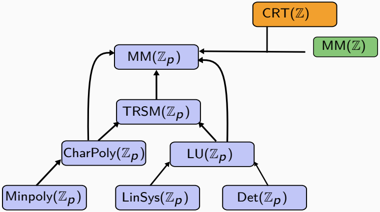
FFLAS-FFPACK: dense linear algebra mod p

What is provided ?

FFLAS-FFPACK: dense linear algebra $\bmod p$

What is provided ?

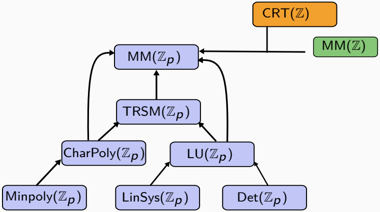
- high-performance matrix multiplication
- tuned reductions to matrix mul : minimizing $\bmod p$/memory



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

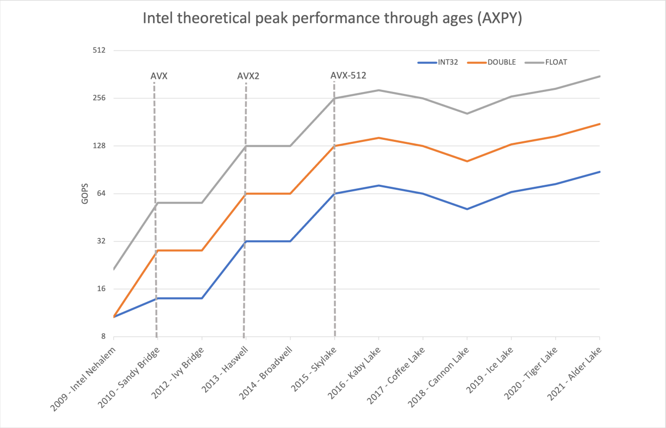
- delegate some optimization to BLAS library: ✓ cache re-use
- subcubic matrix multiplication (Strassen-Winograd)
- generic interface for Intel SIMD intrinsic (SSE/AVX/AVX2/AVX512)
- PALADIn: PArallel Linear Algebra Dedicated Interface

Machine word arithmetic for exact matrix multiplication

Main operation: AXPY: $a \times b + c$

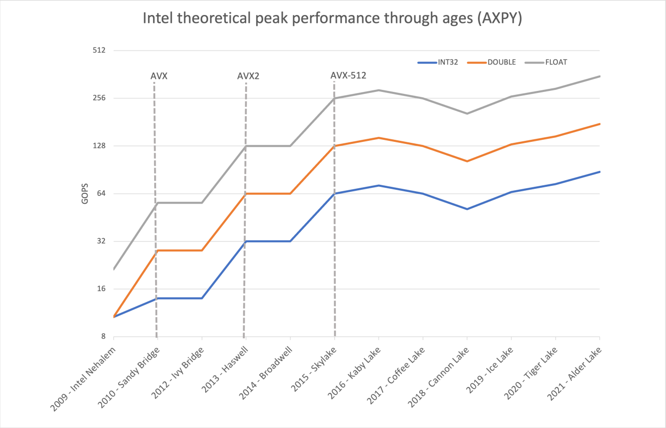
Machine word arithmetic for exact matrix multiplication

Main operation: AXPY: $a \times b + c$



Machine word arithmetic for exact matrix multiplication

Main operation: AXPY: $a \times b + c$



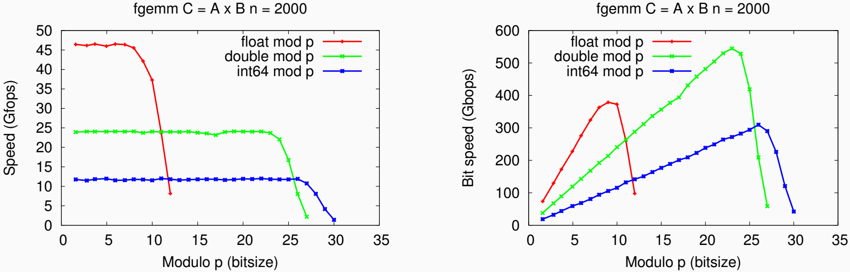
FP numbers seems the good choice !!!

✓ Many BLAS libraries available: OpenBlas, BLIS, MKL, etc.

Machine word arithmetic for matrix multiplication $\bmod p$

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

$\Rightarrow$ limit p to half wordsize

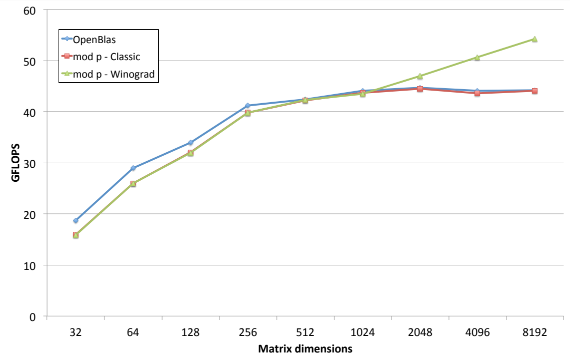


benchmark on Intel Sandy Bridge (courtesy of C. Pernet)

- best performances with FP (except in corner cases)
- double precision delivers highest bit op. throughput

Matrix multiplication $\bmod p$ ($< 26 bits$)

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

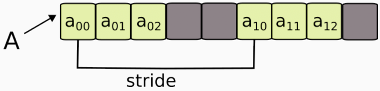


benchmark on Intel Haswell, $p < 20$ bits

FFLAS-FFPACK: API design

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

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



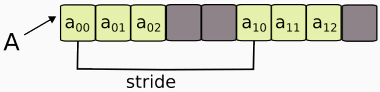
- most optimizations use static type of $\mathbb{Z}_p$ (not the value of p)

  ⇒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)

  ⇒type traits to specialized template functions: `fgemm`, `ftrsm`, etc.

```
Modular<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*B =( 0*C + 1*A*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);
```

Dense linear algebra modulo p: implementation approach

Use algorithmic reduction to `fgemm`

$\Rightarrow$ but minimize modular reductions

Dense linear algebra modulo p: implementation approach

Use algorithmic reduction to `fgemm`

$\Rightarrow$ but minimize modular reductions

Example with ftrsm:

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

- $A_3 X_2 = B_2 \bmod p$
- $D = B_1 - A_2 X_2$ over $\mathbb{Z}$
- $A_1 X_1 = D \bmod p$

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

Dense linear algebra modulo p: implementation approach

Use algorithmic reduction to `fgemm`

$\Rightarrow$ but minimize modular reductions

Example with ftrsm:

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

- $A_3 X_2 = B_2 \bmod p$
- $D = B_1 - A_2 X_2$ over $\mathbb{Z}$
- $A_1 X_1 = D \bmod p$

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

✓ only $O(n^2)$ modular reductions

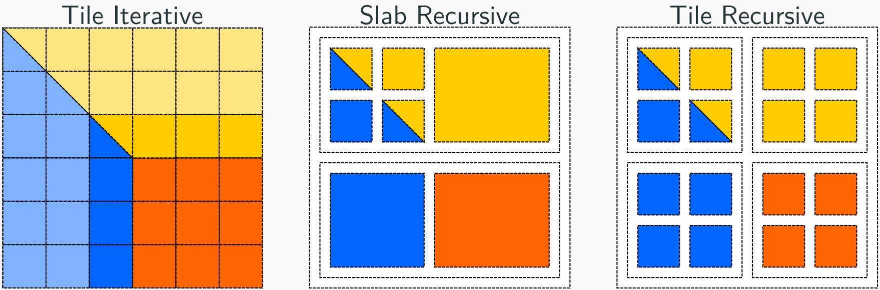
✓ practical performance ~ `fgemm`

Dense linear algebra modulo p: implementation approach

LQUP/PLUQ factorization reduces to `fgemm` and `ftrsm`

Dense linear algebra modulo p: implementation approach

LQUP/PLUQ factorization reduces to `fgemm` and `ftrsm`



`getrf`: $A \to L, U$

`trsm`: $B \leftarrow BU^{-1}$,

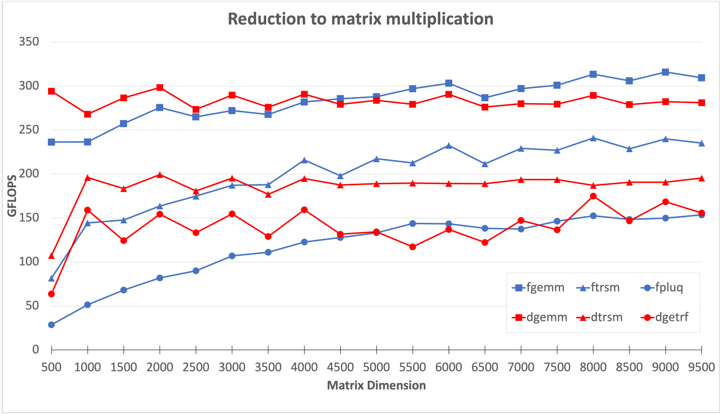
$B \leftarrow L^{-1}B$

`gemm`: $C \leftarrow C - A \times B$

careful choice to

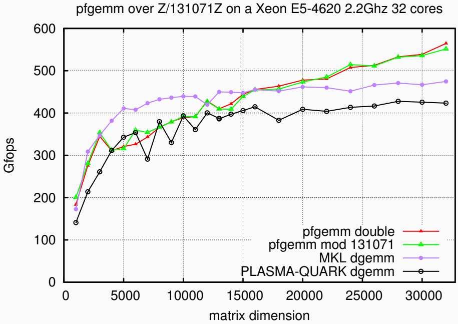
- minimize mod p [Dumas, Pernet, Sultan 13]
- benefit more from Strassen/Winograd

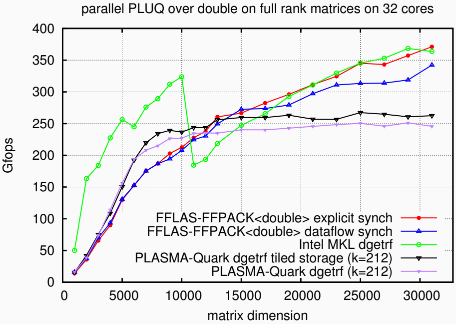
Dense linear algebra modulo p (< 26 bits): reductions in practice



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 $\bmod p$ ($\geq 32 bits$)

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 $\bmod p$ ($\geq 32 bits$)

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. $\bmod q$ (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. $\bmod p_i$ (CRT)
  $\mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \underbrace{\mathbb{Z}_{p_1 \times \cdots \times p_d} \rightarrow \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_d} \rightarrow \mathbb{Z}}_{\text{RNS conversions}} \rightarrow \mathbb{Z}_p$
  $AB \bmod (p_1 \times \cdots \times p_d) \longleftrightarrow (AB \bmod p_1, \ldots, AB \bmod p_d)$

Matrix multiplication $\bmod p$ ($\geq 32 bits$)

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. $\bmod q$ (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. $\bmod p_i$ (CRT)

  $\mathbb{Z} \rightarrow \mathbb{Z}_m \rightarrow \underbrace{\mathbb{Z}_{p_1 \times \cdots \times p_d} \rightarrow \mathbb{Z}_{p_1} \times \cdots \times \mathbb{Z}_{p_d} \rightarrow \mathbb{Z}}_{\text{RNS conversions}} \rightarrow \mathbb{Z}_p$

  $AB \bmod (p_1 \times \cdots \times p_d) \longleftrightarrow (AB \bmod p_1, \ldots, AB \bmod p_d)$

How to improve the reduction ? especially RNS

Optimizing RNS conversions

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]

$\Rightarrow$ hard to optimize in practice

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

$\Rightarrow$ can be reduced to matrix mult. for many conversion [DGLS18]

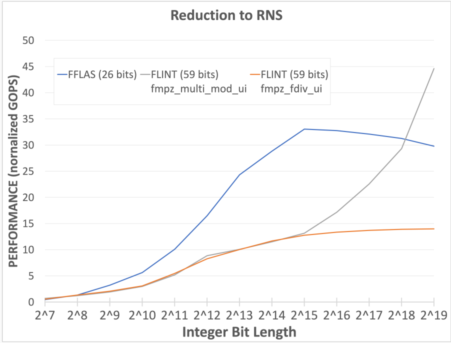
- pseudo-reduction:

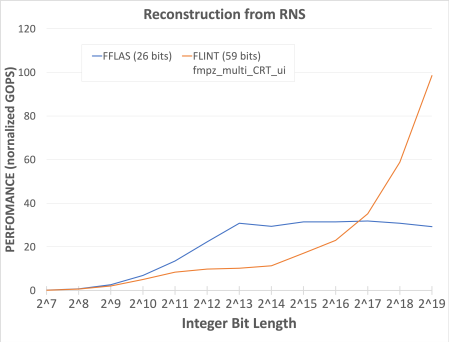
$$A_0 + A_1\beta + \cdots A_{d-1}\beta^{d-1} \longrightarrow \begin{bmatrix} A_0 & \dots & A_{d-1} \end{bmatrix} \times \begin{bmatrix} \beta^i \bmod p_j \end{bmatrix}_{i,j}$$

$$O(d) \longrightarrow O(\log d)$$

- r RNS conversions: $O(\mathbf{r}\mathbf{d}^{\omega-1}) + O(d^2)$ word op.

Simultaneous conversions with RNS: in practice



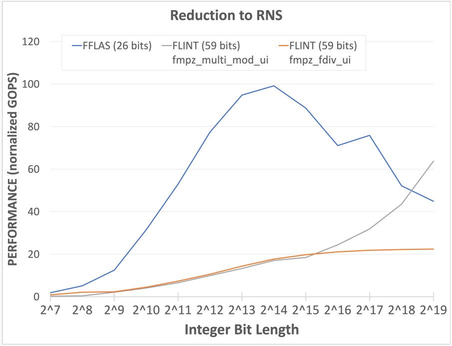


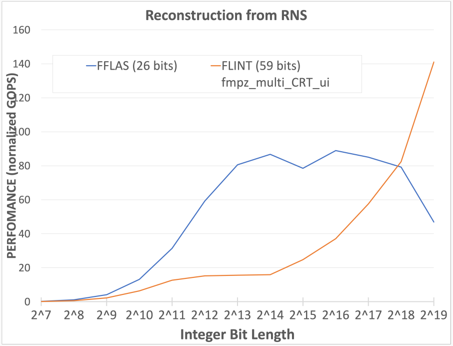
benchmark on Intel Ice Lake - for matrix multiplication ($n = 16$)

One can extend the p_j without sacrifying too much performance

$\Rightarrow$ doubling the prime size halves the number of moduli

Simultaneous conversions with RNS: in practice





benchmark on Apple M1 Max laptop - for matrix multiplication ($n = 16$)

One can extend the p_j without sacrifying too much performance

$\Rightarrow$ doubling the prime size halves the number of moduli

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

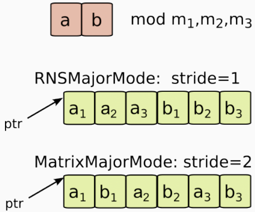
FFLAS: RNS implementation

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
  $\Rightarrow$ contiguous scalar/matrix residues
- redefinition of pointer/iterator
  $\Rightarrow$ handling RNS strides : `ptr+i`, `*ptr`
- fix $\beta = 2^{16}$ and 26-bits moduli



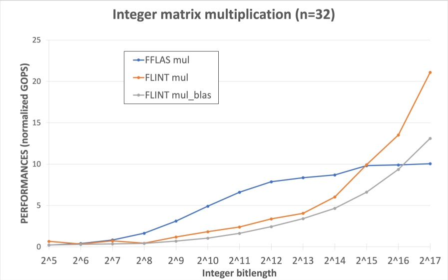
FFLAS integer matrix multiplication

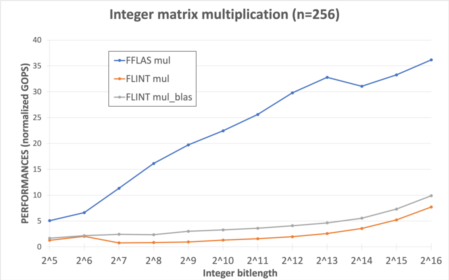
our solution: use multi-modular approach $O(n^{\omega} d + n^2 d^{\omega-1})$
$\hookrightarrow$ reduce everything to `dgemm`

FFLAS integer matrix multiplication

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

$\hookrightarrow$ reduce everything to `dgemm`



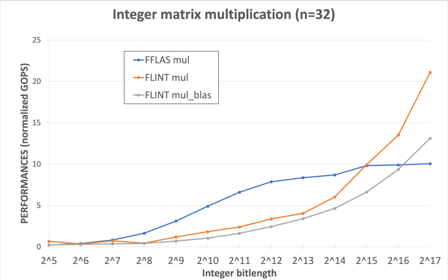


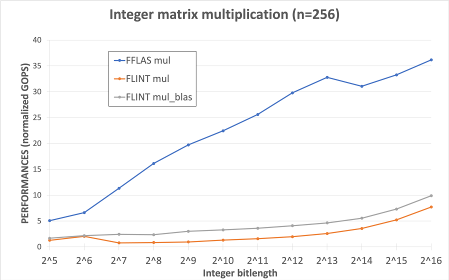
benchmark on Apple M1 Max laptop - 1 core (AMX - 2022)

FFLAS integer matrix multiplication

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

$\hookrightarrow$ reduce everything to `dgemm`





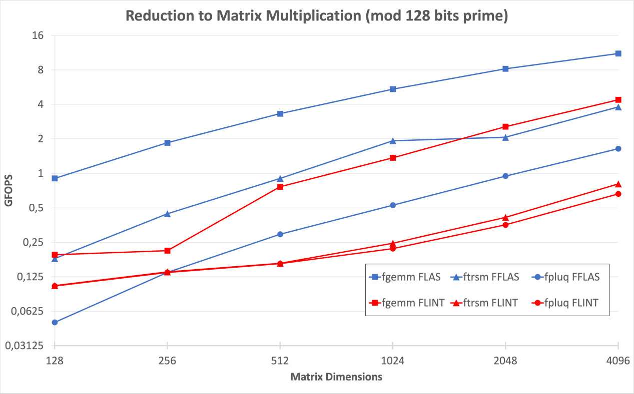
benchmark on Apple M1 Max laptop - 1 core (AMX - 2022)

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

$\Rightarrow$ could be slightly improved by incorporating $\bmod p$ 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



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

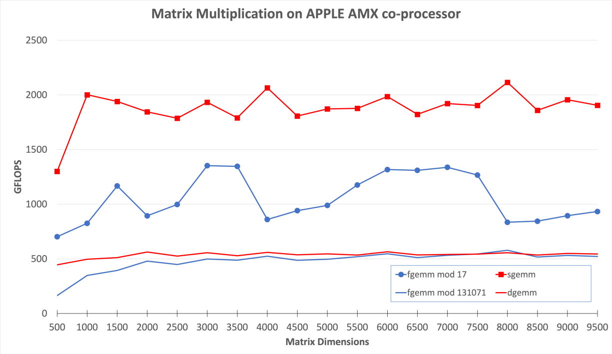
$\Rightarrow$ Timings: 1024×1024 matrices in less than a second

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
  $\Rightarrow$Apple M1 Max processor is $4\times$ 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
  $\Rightarrow$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 $\bmod m_i$ then

$$\begin{bmatrix} |a|_{m_1} \\ \vdots \\ |a|_{m_d} \end{bmatrix} = \begin{bmatrix} 1 & |\beta|_{m_1} & \cdots & |\beta^{d-1}|_{m_1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & |\beta|_{m_d} & \cdots & |\beta^{d-1}|_{m_d} \end{bmatrix} \times \begin{bmatrix} a0 \\ \vdots \\ a_{d-1} \end{bmatrix} - \begin{bmatrix} q_1 m_1 \\ \vdots \\ q_d m_d \end{bmatrix}$$

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
$O(n^2 d^{\omega-1} + n^2 d\mathsf{M}(\log d) + d^2)$ word op.

Simultaneous conversions with RNS: CRT

CRT formulae : $a = \left(\sum_{i=1}^{k} |aM_i^{-1}|_{m_1} \cdot M_i \right) \bmod M$ 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 $a = a_0 + \cdots + a_{k-1}\beta^{k-1} \bmod M$.

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.