

Exact Linear Algebra Algorithmic: Theory and Practice

ISSAC'15 Tutorial

Clément Pernet

Université Grenoble Alpes, Inria, LIP-AriC

July 6, 2015

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size: ▶ integers with a priori bounds

 ▶ $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \dots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: $K[X]$ for K any of the above

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size: ▶ integers with a priori bounds

 ▶ $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \dots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: $K[X]$ for K any of the above

Several implemenations for the same domain: better fits FFT, LinAlg, etc

Exact linear algebra

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

Black-box: no access to the storage, only *apply* to a vector

Coefficient domains:

Word size: ▶ integers with a priori bounds

 ▶ $\mathbb{Z}/p\mathbb{Z}$ for p of ≈ 32 bits

Multi-precision: $\mathbb{Z}/p\mathbb{Z}$ for p of $\approx 100, 200, 1000, 2000, \dots$ bits

Arbitrary precision: $\mathbb{Z}, \mathbb{Q}$

Polynomials: $K[X]$ for K any of the above

Several implemenations for the same domain: better fits FFT, LinAlg, etc

Need to structure the design.

Exact linear algebra

Motivations

Comp. Number Theory:	CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense
Graph Theory:	MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse
Discrete log.:	LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse
Integer Factorization:	NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse
Algebraic Attacks:	Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense
List decoding of RS codes:	Lattice reduction, over $\text{GF}(q)[X]$, Structured

Exact linear algebra

Motivations

Comp. Number Theory:	CharPoly, LinSys, Echelon, over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}$, Dense
Graph Theory:	MatMul, CharPoly, Det, over $\mathbb{Z}$, Sparse
Discrete log.:	LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 120$ bits, Sparse
Integer Factorization:	NullSpace, over $\mathbb{Z}/2\mathbb{Z}$, Sparse
Algebraic Attacks:	Echelon, LinSys, over $\mathbb{Z}/p\mathbb{Z}$, $p \approx 20$ bits, Sparse & Dense
List decoding of RS codes:	Lattice reduction, over $\text{GF}(q)[X]$, Structured

Need for high performance.

The scope of this presentation:

- ▶ not an exhaustive overview on linear algebra algorithmic and complexity improvements
- ▶ a few guidelines, for the use and design of exact linear algebra in practice
- ▶ bridging the theoretical algorithmic development and practical efficiency concerns

Outline

- 1 Choosing the underlying arithmetic
 - Using boolean arithmetic
 - Using machine word arithmetic
 - Larger field sizes
- 2 Reductions and building blocks
 - In dense linear algebra
 - In blackbox linear algebra
- 3 Size dimension trade-offs
 - Hermite normal form
 - Frobenius normal form
- 4 Parallel exact linear algebra
 - Ingredients for the parallelization
 - Parallel dense linear algebra mod p

Outline

- 1 Choosing the underlying arithmetic
 - Using boolean arithmetic
 - Using machine word arithmetic
 - Larger field sizes
- 2 Reductions and building blocks
 - In dense linear algebra
 - In blackbox linear algebra
- 3 Size dimension trade-offs
 - Hermite normal form
 - Frobenius normal form
- 4 Parallel exact linear algebra
 - Ingredients for the parallelization
 - Parallel dense linear algebra mod p

Achieving high practical efficiency

Most of linear algebra operations boil down to (a lot of)

$$y \leftarrow y \pm a * b$$

- ▶ dot-product
- ▶ matrix-matrix multiplication
- ▶ rank 1 update in Gaussian elimination
- ▶ Schur complements, ...

Efficiency relies on

- ▶ fast arithmetic
- ▶ fast memory accesses

Here: focus on dense linear algebra

Which computer arithmetic ?

Many base fields/rings to support

$\mathbb{Z}_2$	1 bit
$\mathbb{Z}_{3,5,7}$	2-3 bits
$\mathbb{Z}_p$	4-26 bits
$\mathbb{Z}, \mathbb{Q}$	> 32 bits
$\mathbb{Z}_p$	> 32 bits

Which computer arithmetic ?

Many base fields/rings to support

$\mathbb{Z}_2$	1 bit
$\mathbb{Z}_{3,5,7}$	2-3 bits
$\mathbb{Z}_p$	4-26 bits
$\mathbb{Z}, \mathbb{Q}$	> 32 bits
$\mathbb{Z}_p$	> 32 bits

Available CPU arithmetic

- ▶ boolean
- ▶ integer (fixed size)
- ▶ floating point
- ▶ .. and their vectorization

Which computer arithmetic ?

Many base fields/rings to support

$\mathbb{Z}_2$	1 bit	↔ bit-packing
$\mathbb{Z}_{3,5,7}$	2-3 bits	↔ bit-slicing, bit-packing
$\mathbb{Z}_p$	4-26 bits	↔ CPU arithmetic
$\mathbb{Z}, \mathbb{Q}$	> 32 bits	↔ multiprec. ints, big ints, CRT, lifting
$\mathbb{Z}_p$	> 32 bits	↔ multiprec. ints, big ints, CRT

Available CPU arithmetic

- ▶ boolean
- ▶ integer (fixed size)
- ▶ floating point
- ▶ .. and their vectorization

Which computer arithmetic ?

Many base fields/rings to support

$\mathbb{Z}_2$	1 bit	$\rightsquigarrow$ bit-packing
$\mathbb{Z}_{3,5,7}$	2-3 bits	$\rightsquigarrow$ bit-slicing, bit-packing
$\mathbb{Z}_p$	4-26 bits	$\rightsquigarrow$ CPU arithmetic
$\mathbb{Z}, \mathbb{Q}$	> 32 bits	$\rightsquigarrow$ multiprec. ints, big ints, CRT, lifting
$\mathbb{Z}_p$	> 32 bits	$\rightsquigarrow$ multiprec. ints, big ints, CRT
$\text{GF}(p^k) \equiv \mathbb{Z}_p[X]/(Q)$		$\rightsquigarrow$ Polynomial, Kronecker, Zech log, ...

Available CPU arithmetic

- ▶ boolean
- ▶ integer (fixed size)
- ▶ floating point
- ▶ .. and their vectorization

Dense linear algebra over $\mathbb{Z}_2$: bit-packing

`uint64_t` $\equiv (\mathbb{Z}_2)^{64} \rightsquigarrow$

$\wedge$: bit-wise XOR, $(+ \bmod 2)$

$\&$: bit-wise AND, $(* \bmod 2)$

dot-product (a,b)

```
uint64_t t = 0;
for (int k=0; k < N/64; ++k)
    t ^= a[k] & b[k];
c = parity(t)
```

parity(x)

```
if (size(x) == 1)
    return x;
else return parity (High(x) ^ Low(x))
```

$\rightsquigarrow$ Can be parallelized on 64 instances.

Tabulation:

- ▶ avoid computing parities
- ▶ balance computation vs communication
- ▶ (slight) complexity improvement possible

Tabulation:

- ▶ avoid computing parities
- ▶ balance computation vs communication
- ▶ (slight) complexity improvement possible

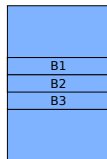
The Four Russian method [Arlazarov, Dinic, Kronrod, Faradzev 70]

- 1 compute all 2^k linear combinations of k rows of B .

Gray code: each new line costs 1 vector add (vs $k/2$)

- 2 multiply chunks of length k of A by table look-up

	1	0	1	
	1	1	1	
	1	0	0	
	1	0	1	
	0	0	1	
	1	0	1	



0	0	0	0
B3	0	0	1
B2+B3	0	1	1
B2	0	1	0
B1+B2	1	1	0
B1+B2+B3	1	1	1
B1+B3	1	0	1
B1	1	0	0



Tabulation:

- ▶ avoid computing parities
- ▶ **balance computation vs communication**
- ▶ **(slight) complexity improvement possible**

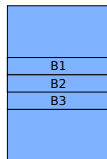
The Four Russian method [Arlazarov, Dinic, Kronrod, Faradzev 70]

- 1 compute all 2^k linear combinations of k rows of B .

Gray code: each new line costs 1 vector add (vs $k/2$)

- 2 multiply chunks of length k of A by table look-up

	1	0	1	
	1	1	1	
	1	0	0	
	1	0	1	
	0	0	1	
	1	0	1	



0	0	0	0
B3	0	0	1
B2+B3	0	1	1
B2	0	1	0
B1+B2	1	1	0
B1+B2+B3	1	1	1
B1+B3	1	0	1
B1	1	0	0

- ▶ **For** $k = \log n \rightsquigarrow O(n^3 / \log n)$.
- ▶ **In practice:** choice of k s.t. the table fits in L2 cache.

Dense linear algebra over $\mathbb{Z}_2$

The M4RI library [Albrecht Bard Hart 10]

- ▶ bit-packing
- ▶ Method of the Four Russians
- ▶ SIMD vectorization of boolean instructions (128 bits registers)
- ▶ Cache optimization
- ▶ Strassen's $O(n^{2.81})$ algorithm

n	7000	14 000	28 000
SIMD bool arithmetic	2.109s	15.383s	111.82
SIMD + 4 Russians	0.256s	2.829s	29.28s
SIMD + 4 Russians + Strassen	0.257s	2.001s	15.73

Table : Matrix product $n \times n$ by $n \times n$, on an i5 SandyBridge 2.6Ghz.

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\}$$

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\mathbb{Z}_3 = \{0, 1, -1\} = \{00, 01, 10\} \rightsquigarrow \text{add/sub in 7 bool ops}$$

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\begin{aligned}\mathbb{Z}_3 = \{0, 1, -1\} &= \{00, 01, 10\} && \rightsquigarrow \text{add/sub in 7 bool ops} \\ &= \{00, 10, 11\} && \rightsquigarrow \text{add/sub in 6 bool ops}\end{aligned}$$

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\begin{aligned}\mathbb{Z}_3 = \{0, 1, -1\} &= \{00, 01, 10\} \rightsquigarrow \text{add/sub in 7 bool ops} \\ &= \{00, 10, 11\} \rightsquigarrow \text{add/sub in 6 bool ops}\end{aligned}$$

Bit-slicing

$$(-1, 0, 1, 0, 1, -1, -1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)$$

Stored as 2 words

$$\begin{aligned}&(1, 0, 1, 0, 1, 1, 0) \\ &(1, 0, 0, 0, 0, 1, 0)\end{aligned}$$

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\begin{aligned}\mathbb{Z}_3 = \{0, 1, -1\} &= \{00, 01, 10\} \rightsquigarrow \text{add/sub in 7 bool ops} \\ &= \{00, 10, 11\} \rightsquigarrow \text{add/sub in 6 bool ops}\end{aligned}$$

Bit-slicing

$$(-1, 0, 1, 0, 1, -1, -1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)$$

Stored as 2 words

$$\begin{pmatrix} 1, 0, 1, 0, 1, 1, 0 \\ 1, 0, 0, 0, 0, 1, 0 \end{pmatrix}$$

$$\rightsquigarrow \vec{y} \leftarrow \vec{y} + x\vec{b} \text{ for } x \in \mathbb{Z}_3, \vec{y}, \vec{b} \in \mathbb{Z}_3^{64} \text{ in 6 boolean word ops.}$$

Dense linear algebra over $\mathbb{Z}_3, \mathbb{Z}_5$ [Boothby & Bradshaw 09]

$$\begin{aligned}\mathbb{Z}_3 = \{0, 1, -1\} &= \{00, 01, 10\} \rightsquigarrow \text{add/sub in 7 bool ops} \\ &= \{00, 10, 11\} \rightsquigarrow \text{add/sub in 6 bool ops}\end{aligned}$$

Bit-slicing

$$(-1, 0, 1, 0, 1, -1, -1, 0) \in \mathbb{Z}_3^8 \rightarrow (11, 00, 10, 00, 10, 11, 00)$$

Stored as 2 words

$$\begin{aligned}&(1, 0, 1, 0, 1, 1, 0) \\ &(1, 0, 0, 0, 0, 1, 0)\end{aligned}$$

$$\rightsquigarrow \vec{y} \leftarrow \vec{y} + x\vec{b} \text{ for } x \in \mathbb{Z}_3, \vec{y}, \vec{b} \in \mathbb{Z}_3^{64} \text{ in 6 boolean word ops.}$$

Recipe for $\mathbb{Z}_5$

- ▶ Use redundant representations on 3 bits + bit-slicing
- ▶ integer add + bool operations
- ▶ Pseudo-reduction mod 5 ($4 \rightarrow 3$ bits) in 8 bool ops found by computer assisted search.

Dense linear algebra over $\mathbb{Z}_p$ for word-size p

Delayed modular reductions

- 1 Compute using integer arithmetic
- 2 Reduce modulo p only when necessary

Dense linear algebra over $\mathbb{Z}_p$ for word-size p

Delayed modular reductions

- ① Compute using integer arithmetic
- ② Reduce modulo p only when necessary

When to reduce ?

Bound the values of all intermediate computations.

► A priori:

Representation of $\mathbb{Z}_p$	$\{0 \dots p-1\}$	$\{-\frac{p-1}{2} \dots \frac{p-1}{2}\}$
Scalar product, Classic MatMul	$n(p-1)^2$	$n \left(\frac{p-1}{2}\right)^2$

Dense linear algebra over $\mathbb{Z}_p$ for word-size p

Delayed modular reductions

- ① Compute using integer arithmetic
- ② Reduce modulo p only when necessary

When to reduce ?

Bound the values of all intermediate computations.

- A priori:

Representation of $\mathbb{Z}_p$	$\{0 \dots p-1\}$	$\{-\frac{p-1}{2} \dots \frac{p-1}{2}\}$
Scalar product, Classic MatMul	$n(p-1)^2$	$n \left(\frac{p-1}{2}\right)^2$
Strassen-Winograd MatMul (ℓ rec. levels)	$\left(\frac{1+3^\ell}{2}\right)^2 \lfloor \frac{n}{2^\ell} \rfloor (p-1)^2$	$9^\ell \lfloor \frac{n}{2^\ell} \rfloor \left(\frac{p-1}{2}\right)^2$

Dense linear algebra over $\mathbb{Z}_p$ for word-size p

Delayed modular reductions

- ① Compute using integer arithmetic
- ② Reduce modulo p only when necessary

When to reduce ?

Bound the values of all intermediate computations.

- ▶ A priori:

Representation of $\mathbb{Z}_p$	$\{0 \dots p-1\}$	$\{-\frac{p-1}{2} \dots \frac{p-1}{2}\}$
Scalar product, Classic MatMul	$n(p-1)^2$	$n \left(\frac{p-1}{2}\right)^2$
Strassen-Winograd MatMul (ℓ rec. levels)	$\left(\frac{1+3^\ell}{2}\right)^2 \lfloor \frac{n}{2^\ell} \rfloor (p-1)^2$	$9^\ell \lfloor \frac{n}{2^\ell} \rfloor \left(\frac{p-1}{2}\right)^2$

- ▶ Maintain locally a bounding interval on all matrices computed

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

- 1 use CPU's **integer arithmetic units**

$y \leftarrow y + a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

1 use CPU's **integer arithmetic units**

$y \leftarrow y + a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

```
movq    (%rax,%rdx,8), %rax
```

```
imulq   -56(%rbp), %rax
```

```
addq    %rax, %rcx
```

```
movq    -80(%rbp), %rax
```


Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

- 1 use CPU's **integer arithmetic units** + vectorization

$y += a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

```
movq    (%rax,%rdx,8), %rax
```

```
imulq   -56(%rbp), %rax
```

```
addq    %rax, %rcx
```

```
movq    -80(%rbp), %rax
```

```
vpmuludq %xmm3, %xmm0,%xmm0
```

```
vpaddq   %xmm2,%xmm0,%xmm0
```

```
vpsllq   $32,%xmm0,%xmm0
```

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

- 1 use CPU's **integer arithmetic units** + vectorization

$y += a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

```
movq    (%rax,%rdx,8), %rax
```

```
imulq   -56(%rbp), %rax
```

```
addq    %rax, %rcx
```

```
movq    -80(%rbp), %rax
```

```
vpmuludq %xmm3, %xmm0, %xmm0
```

```
vpaddq   %xmm2, %xmm0, %xmm0
```

```
vpsllq   $32, %xmm0, %xmm0
```

- 2 use CPU's **floating point units**

$y += a * b$: correct if $|ab + y| < 2^{53} \rightsquigarrow |a|, |b| < 2^{26}$

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

- 1 use CPU's **integer arithmetic units** + vectorization

$y += a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

```
movq    (%rax,%rdx,8), %rax
```

```
imulq   -56(%rbp), %rax
```

```
addq    %rax, %rcx
```

```
movq    -80(%rbp), %rax
```

```
vpmuludq %xmm3, %xmm0, %xmm0
```

```
vpaddq   %xmm2, %xmm0, %xmm0
```

```
vpsllq   $32, %xmm0, %xmm0
```

- 2 use CPU's **floating point units**

$y += a * b$: correct if $|ab + y| < 2^{53} \rightsquigarrow |a|, |b| < 2^{26}$

```
movsd    (%rax,%rdx,8), %xmm0
```

```
mulsd    -56(%rbp), %xmm0
```

```
addsd    %xmm0, %xmm1
```

```
movq     %xmm1, %rax
```

Computing over fixed size integers

How to compute with (machine word size) integers efficiently?

- 1 use CPU's **integer arithmetic units** + vectorization

$y += a * b$: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

```
movq    (%rax,%rdx,8), %rax
```

```
imulq   -56(%rbp), %rax
```

```
addq    %rax, %rcx
```

```
movq    -80(%rbp), %rax
```

```
vpmuludq %xmm3, %xmm0, %xmm0
```

```
vpaddq   %xmm2, %xmm0, %xmm0
```

```
vpsllq   $32, %xmm0, %xmm0
```

- 2 use CPU's **floating point units** + vectorization

$y += a * b$: correct if $|ab + y| < 2^{53} \rightsquigarrow |a|, |b| < 2^{26}$

```
movsd   (%rax,%rdx,8), %xmm0
```

```
mulsd   -56(%rbp), %xmm0
```

```
addsd   %xmm0, %xmm1
```

```
movq    %xmm1, %rax
```

```
vinserftf128 $0x1, 16(%rcx,%rax), %ymm0,
```

```
vmulpd   %ymm1, %ymm0, %ymm0
```

```
vaddpd   (%rsi,%rax), %ymm0, %ymm0
```

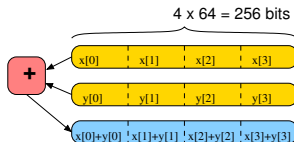
```
vmovapd  %ymm0, (%rsi,%rax)
```

Exploiting *in-core* parallelism

Ingredients

SIMD: Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

MMX	64 bits
SSE	128bits
AVX	256 bits
AVX-512	512 bits

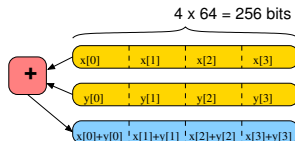


Exploiting *in-core* parallelism

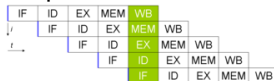
Ingredients

SIMD: Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

MMX	64 bits
SSE	128bits
AVX	256 bits
AVX-512	512 bits



Pipeline: amortize the latency of an operation when used repeatedly
throughput of 1 op/ Cycle for all
arithmetic ops considered here

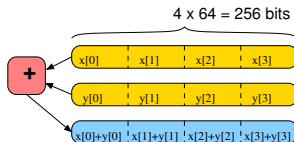


Exploiting *in-core* parallelism

Ingredients

SIMD: Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

MMX	64 bits
SSE	128bits
AVX	256 bits
AVX-512	512 bits



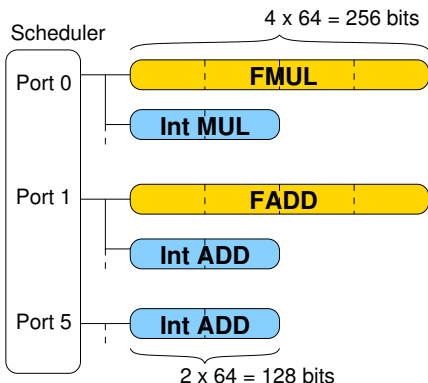
Pipeline: amortize the latency of an operation when used repeatedly
throughput of 1 op/ Cycle for all
arithmetic ops considered here



Execution Unit parallelism: multiple arith. units acting simultaneously on
distinct registers

SIMD and vectorization

Intel Sandybridge micro-architecture



Performs at every clock cycle:

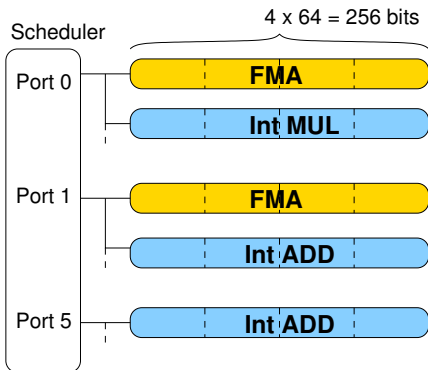
- ▶ 1 Floating Pt. Mul $\times 4$
- ▶ 1 Floating Pt. Add $\times 4$

Or:

- ▶ 1 Integer Mul $\times 2$
- ▶ 2 Integer Add $\times 2$

SIMD and vectorization

Intel Haswell micro-architecture



Performs at every clock cycle:

- ▶ 2 Floating Pt. Mul & Add $\times 4$

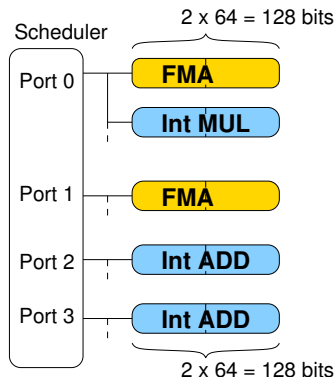
Or:

- ▶ 1 Integer Mul $\times 4$
- ▶ 2 Integer Add $\times 4$

FMA: Fused Multiplying & Accumulate, $c += a * b$

SIMD and vectorization

AMD Bulldozer micro-architecture



Performs at every clock cycle:

- ▶ 2 Floating Pt. Mul & Add $\times 2$

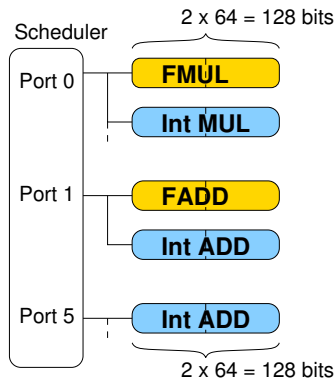
Or:

- ▶ 1 Integer Mul $\times 2$
- ▶ 2 Integer Add $\times 2$

FMA: Fused Multiplying & Accumulate, $c += a * b$

SIMD and vectorization

Intel Nehalem micro-architecture



Performs at every clock cycle:

- ▶ 1 Floating Pt. Mul $\times 2$
- ▶ 1 Floating Pt. Add $\times 2$

Or:

- ▶ 1 Integer Mul $\times 2$
- ▶ 2 Integer Add $\times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	
	FP	256			2	8	3.5	56	
Intel Sandybridge AVX1	INT								
	FP								
AMD Bulldozer FMA4	INT								
	FP								
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT								
	FP								
AMD Bulldozer FMA4	INT								
	FP								
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	
	FP	256	1	1		4	3.3	26.4	
AMD Bulldozer FMA4	INT								
	FP								
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT								
	FP								
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	
	FP	128			2	4	2.1	16.8	
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	6.44
	FP	128			2	4	2.1	16.8	13.1
Intel Nehalem SSE4	INT								
	FP								
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	6.44
	FP	128			2	4	2.1	16.8	13.1
Intel Nehalem SSE4	INT	128	2	1		2	2.66	10.6	
	FP	128	1	1		2	2.66	10.6	
AMD K10 SSE4a	INT								
	FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	6.44
	FP	128			2	4	2.1	16.8	13.1
Intel Nehalem SSE4	INT	128	2	1		2	2.66	10.6	4.47
	FP	128	1	1		2	2.66	10.6	9.6
AMD K10 SSE4a	INT FP								

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	6.44
	FP	128			2	4	2.1	16.8	13.1
Intel Nehalem SSE4	INT	128	2	1		2	2.66	10.6	4.47
	FP	128	1	1		2	2.66	10.6	9.6
AMD K10 SSE4a	INT	64	2	1		1	2.4	4.8	
	FP	128	1	1		2	2.4	9.6	

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# daxpy / Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell AVX2	INT	256	2	1		4	3.5	28	23.3
	FP	256			2	8	3.5	56	49.2
Intel Sandybridge AVX1	INT	128	2	1		2	3.3	13.2	12.1
	FP	256	1	1		4	3.3	26.4	24.6
AMD Bulldozer FMA4	INT	128	2	1		2	2.1	8.4	6.44
	FP	128			2	4	2.1	16.8	13.1
Intel Nehalem SSE4	INT	128	2	1		2	2.66	10.6	4.47
	FP	128	1	1		2	2.66	10.6	9.6
AMD K10 SSE4a	INT	64	2	1		1	2.4	4.8	
	FP	128	1	1		2	2.4	9.6	8.93

Speed of light: $\text{CPU freq} \times (\# \text{ daxpy} / \text{Cycle}) \times 2$

Computing over fixed size integers: ressources

Micro-architecture bible: Agner Fog's software optimization resources
[www.agner.org/optimize]

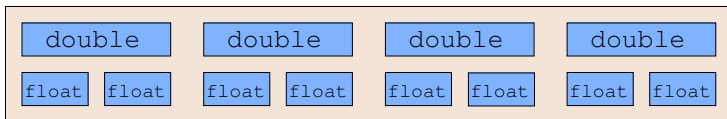
Experiments:

`dgemm (double)`: OpenBLAS [<http://www.openblas.net/>]

`igemm (int64_t)`: Eigen [<http://eigen.tuxfamily.org/>] &
FFLAS-FFPACK [linalg.org/projects/fflas-ffpack]

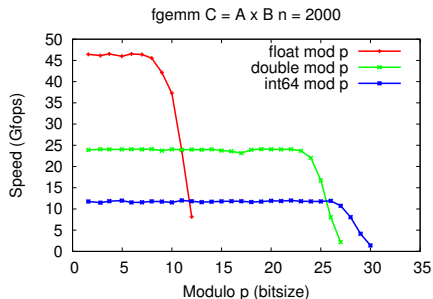
Integer Packing

32 bits: half the precision twice the speed



Gfops	double	float	int64_t	int32_t
Intel SandyBridge	24.7	49.1	12.1	24.7
Intel Haswell	49.2	77.6	23.3	27.4
AMD Bulldozer	13.0	20.7	6.63	11.8

Computing over fixed size integers

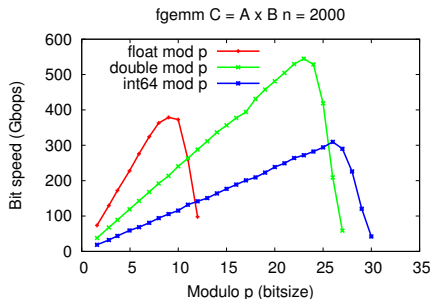
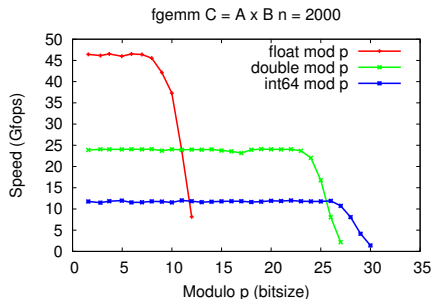


SandyBridge i5-3320M@3.3Ghz. $n = 2000$.

Take home message

- ▶ Floating pt. arith. delivers the highest speed (except in corner cases)
- ▶ 32 bits twice as fast as 64 bits

Computing over fixed size integers



SandyBridge i5-3320M@3.3Ghz. $n = 2000$.

Take home message

- ▶ Floating pt. arith. delivers the highest speed (except in corner cases)
- ▶ 32 bits twice as fast as 64 bits
- ▶ best bit computation throughput for double precision floating points.

Larger finite fields: $\log_2 p \geq 32$

As before:

- 1 Use adequate integer arithmetic
- 2 reduce modulo p only when necessary

Which integer arithmetic?

- 1 big integers (GMP)
- 2 fixed size multiprecision (Givaro-RecInt)
- 3 Residue Number Systems (Chinese Remainder theorem)
 $\rightsquigarrow$ using moduli delivering optimum bitspeed

Larger finite fields: $\log_2 p \geq 32$

As before:

- ① Use adequate integer arithmetic
- ② reduce modulo p only when necessary

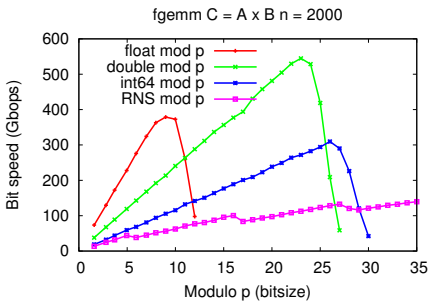
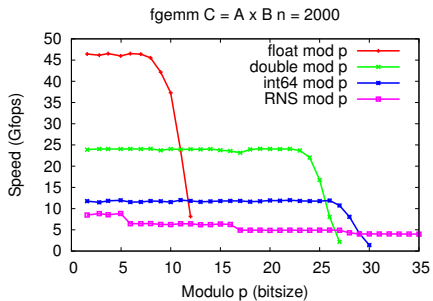
Which integer arithmetic?

- ① big integers (GMP)
- ② fixed size multiprecision (Givaro-RecInt)
- ③ Residue Number Systems (Chinese Remainder theorem)
 $\rightsquigarrow$ using moduli delivering optimum bitspeed

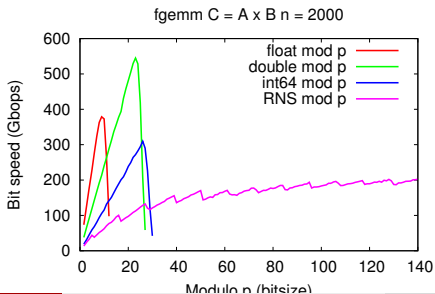
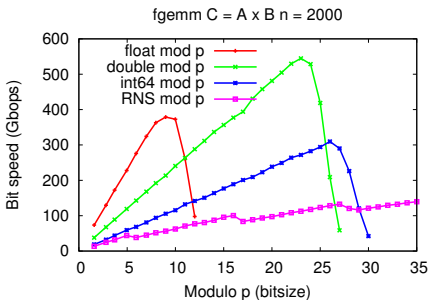
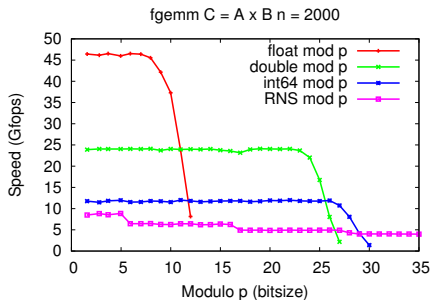
$\log_2 p$	50	100	150
GMP	58.1s	94.6s	140s
RecInt	5.7s	28.6s	837s
RNS	0.785s	1.42s	1.88s

$n = 1000$, on an Intel SandyBridge.

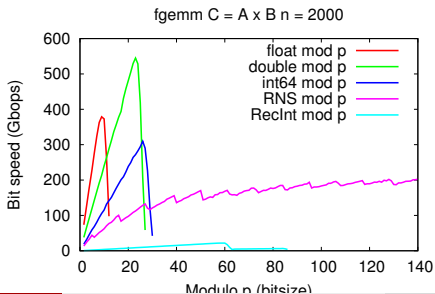
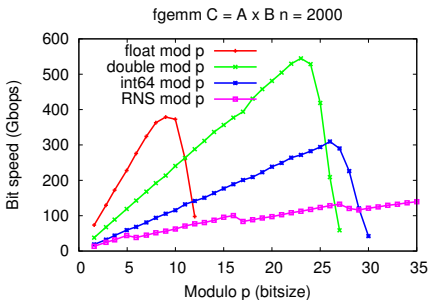
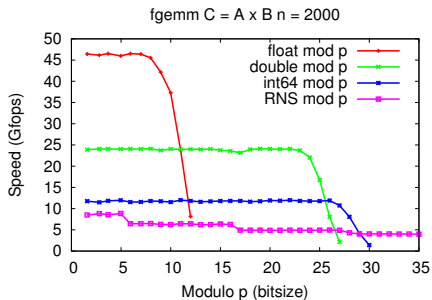
In practice



In practice



In practice



Outline

- 1 Choosing the underlying arithmetic
 - Using boolean arithmetic
 - Using machine word arithmetic
 - Larger field sizes
- 2 Reductions and building blocks
 - In dense linear algebra
 - In blackbox linear algebra
- 3 Size dimension trade-offs
 - Hermite normal form
 - Frobenius normal form
- 4 Parallel exact linear algebra
 - Ingredients for the parallelization
 - Parallel dense linear algebra mod p

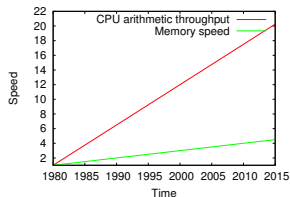
Reductions to building blocks

Huge number of algorithmic variants for a given computation in $O(n^3)$.
Need to structure the design of set of routines :

- ▶ Focus tuning effort on a single routine
- ▶ Some operations deliver better efficiency:
 - ▷ in practice: memory access pattern
 - ▷ in theory: better algorithms

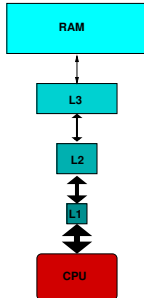
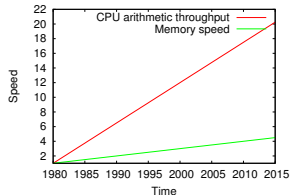
Memory access pattern

- ▶ **The memory wall:** communication speed improves slower than arithmetic



Memory access pattern

- ▶ **The memory wall:** communication speed improves slower than arithmetic
- ▶ Deep memory hierarchy



Memory access pattern

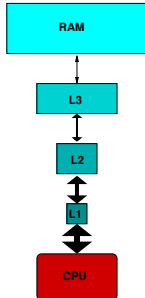
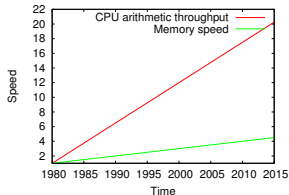
- ▶ **The memory wall:** communication speed improves slower than arithmetic
- ▶ Deep memory hierarchy

⇒ Need to overlap communications by computation

Design of BLAS 3 [Dongarra & Al. 87]

- ▶ Group all ops in **Matrix products** `gemm`:
Work $O(n^3) \gg$ Data $O(n^2)$

MatMul has become a building block in practice



Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication $\rightsquigarrow O(n^\omega)$

[Strassen 69]: $O(n^{2.807})$

$\vdots$

[Schönhage 81] $O(n^{2.52})$

$\vdots$

[Coppersmith, Winograd 90] $O(n^{2.375})$

$\vdots$

[Le Gall 14] $O(n^{2.3728639})$

Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication $\rightsquigarrow O(n^\omega)$

[Strassen 69]:	$O(n^{2.807})$
⋮	
[Schönhage 81]	$O(n^{2.52})$
⋮	
[Coppersmith, Winograd 90]	$O(n^{2.375})$
⋮	
[Le Gall 14]	$O(n^{2.3728639})$

Other operations

[Strassen 69]:	Inverse in $O(n^\omega)$
[Schönhage 72]:	QR in $O(n^\omega)$
[Bunch, Hopcroft 74]:	LU in $O(n^\omega)$
[Ibarra & al. 82]:	Rank in $O(n^\omega)$
[Keller-Gehrig 85]:	CharPoly in $O(n^\omega \log n)$

Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskii, etc)

Matrix Multiplication $\rightsquigarrow O(n^\omega)$

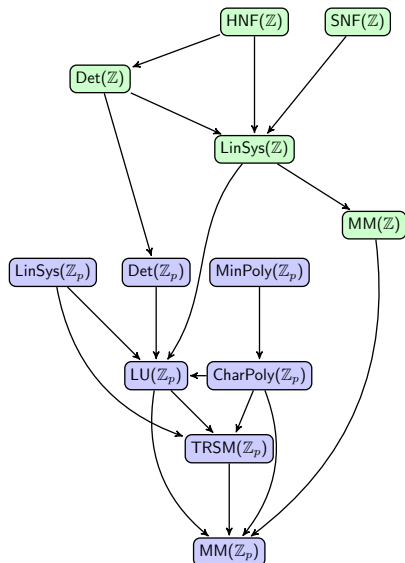
[Strassen 69]:	$O(n^{2.807})$
⋮	
[Schönhage 81]	$O(n^{2.52})$
⋮	
[Coppersmith, Winograd 90]	$O(n^{2.375})$
⋮	
[Le Gall 14]	$O(n^{2.3728639})$

Other operations

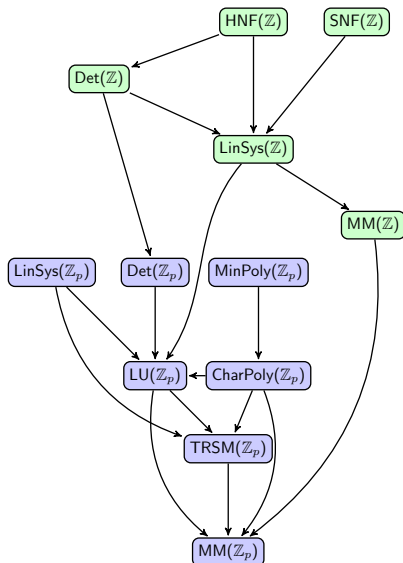
[Strassen 69]:	Inverse in $O(n^\omega)$
[Schönhage 72]:	QR in $O(n^\omega)$
[Bunch, Hopcroft 74]:	LU in $O(n^\omega)$
[Ibarra & al. 82]:	Rank in $O(n^\omega)$
[Keller-Gehrig 85]:	CharPoly in $O(n^\omega \log n)$

MatMul has become a building block in theoretical reductions

Reductions: theory



Reductions: theory

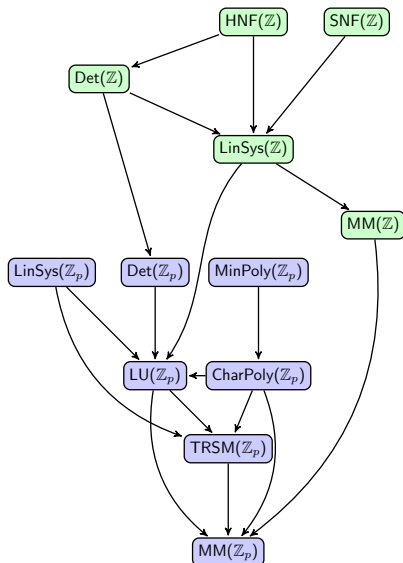


Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Reductions: theory and practice



Common mistrust

Fast linear algebra is

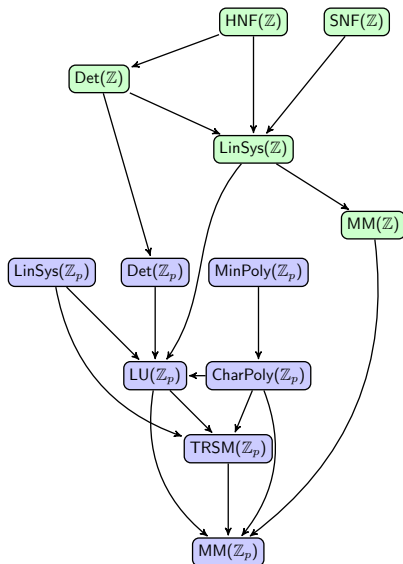
- ✗ never faster
- ✗ numerically unstable

Lucky coincidence

- ✓ same building block **in theory** and **in practice**

⇒ reduction trees are still relevant

Reductions: theory and practice



Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Lucky coincidence

- ✓ same building block **in theory** and **in practice**

⇒ reduction trees are still relevant

Road map towards efficiency in practice

- 1 Tune the MatMul building block.
- 2 Tune the reductions.

Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [FFLAS-FFPACK library]

- Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow k \left(\frac{p-1}{2} \right)^2 < 2^{\text{mantissa}}$$

Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [FFLAS-FFPACK library]

- ▶ Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow k \left(\frac{p-1}{2} \right)^2 < 2^{53}$$

- ▶ Fastest integer arithmetic: double
- ▶ Cache optimizations

$$\rightsquigarrow \text{numerical BLAS}$$

Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [FFLAS-FFPACK library]

- ▶ Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2} \right)^2 < 2^{53}$$

- ▶ Fastest integer arithmetic: double
- ▶ Cache optimizations

$$\rightsquigarrow \text{numerical BLAS}$$

- ▶ Strassen-Winograd $6n^{2.807} + \dots$

Putting it together: MatMul building block over $\mathbb{Z}/p\mathbb{Z}$

Ingredients [FFLAS-FFPACK library]

- ▶ Compute over $\mathbb{Z}$ and delay modular reductions

$$\rightsquigarrow 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2} \right)^2 < 2^{53}$$

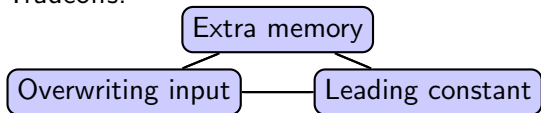
- ▶ Fastest integer arithmetic: double
- ▶ Cache optimizations

$$\rightsquigarrow \text{numerical BLAS}$$

- ▶ Strassen-Winograd $6n^{2.807} + \dots$

with memory efficient schedules [Boyer, Dumas, P. and Zhou 09]

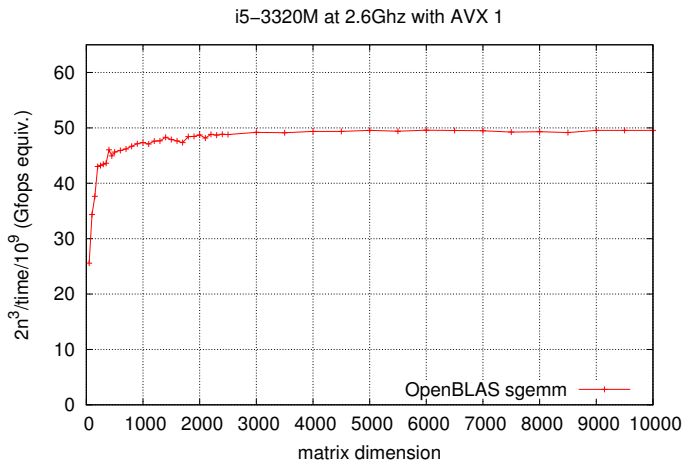
Tradeoffs:



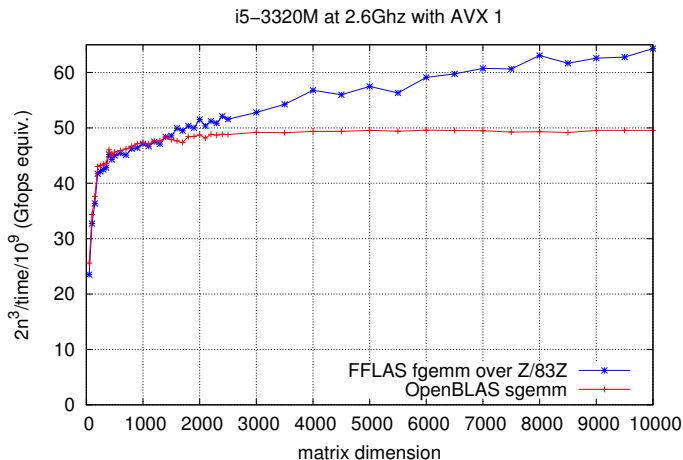
Fully in-place in

$$7.2n^{2.807} + \dots$$

Sequential Matrix Multiplication

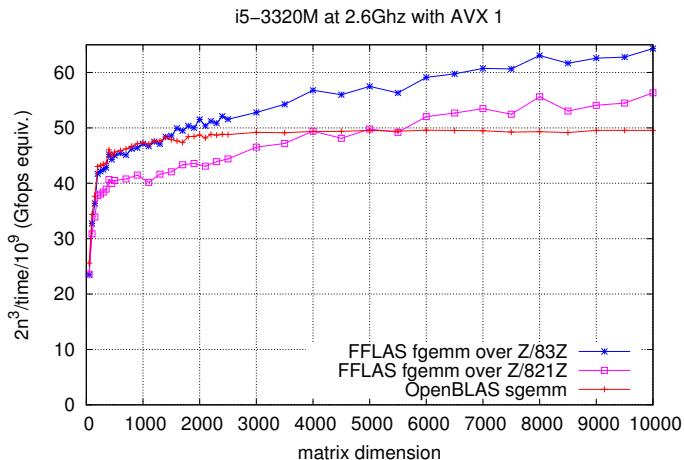


Sequential Matrix Multiplication



$p = 83, \rightsquigarrow 1 \bmod / 10000$ mul.

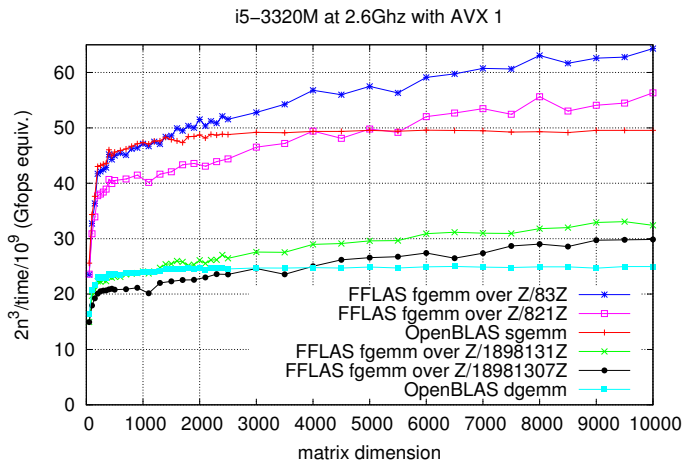
Sequential Matrix Multiplication



$p = 83, \rightsquigarrow 1 \bmod / 10000 \text{ mul.}$

$p = 821, \rightsquigarrow 1 \bmod / 100 \text{ mul.}$

Sequential Matrix Multiplication



$p = 83, \rightsquigarrow 1 \bmod / 10000$ mul.

$p = 821, \rightsquigarrow 1 \bmod / 100$ mul.

$p = 1898131, \rightsquigarrow 1 \bmod / 10000$ mul.

$p = 18981307, \rightsquigarrow 1 \bmod / 100$ mul.

Reductions in dense linear algebra

LU decomposition

- ▶ Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow O(n^\omega)$

n	1000	5000	10000	15000	20000
LAPACK-dgetrf	0.024s	2.01s	14.88s	48.78s	113.66
fflas-ffpack	0.058s	2.46s	16.08s	47.47s	105.96s

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Reductions in dense linear algebra

LU decomposition

- ▶ Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow O(n^\omega)$

n	1000	5000	10000	15000	20000
LAPACK-dgetrf	0.024s	2.01s	14.88s	48.78s	113.66
fflas-ffpack	0.058s	2.46s	16.08s	47.47s	105.96s

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

Characteristic Polynomial

- ▶ A new reduction to matrix multiplication in $O(n^\omega)$.

n	1000	2000	5000	10000
magma-v2.19-9	1.38s	24.28s	332.7s	2497s
fflas-ffpack	0.532s	2.936s	32.71s	219.2s

Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

Reductions in dense linear algebra

LU decomposition

- Block recursive algorithm $\rightsquigarrow$ reduces to MatMul $\rightsquigarrow O(n^\omega)$

n	1000	5000	10000	15000	20000
LAPACK-dgetrf	0.024s	2.01s	14.88s	48.78s	113.66
fflas-ffpack	0.058s	2.46s	16.08s	47.47s	105.96s

Intel Haswell E3-1270 3.0Ghz using OpenBLAS-0.2.9

$\times 7.63$

$\times 6.59$

Characteristic Polynomial

- A new reduction to matrix multiplication in $O(n^\omega)$.

n	1000	2000	5000	10000
magma-v2.19-9	1.38s	24.28s	332.7s	2497s
fflas-ffpack	0.532s	2.936s	32.71s	219.2s

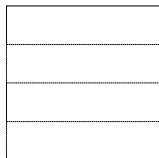
Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

$\times 7.5$

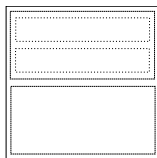
$\times 6.7$

The case of Gaussian elimination

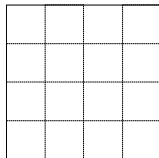
Which reduction to MatMul ?



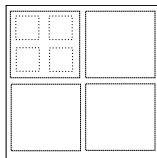
Slab iterative
LAPACK



Slab recursive
FFLAS-FFPACK



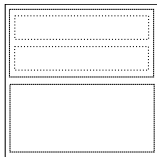
Tile iterative
PLASMA



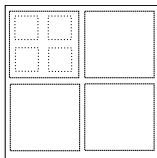
Tile recursive
FFLAS-FFPACK

The case of Gaussian elimination

Which reduction to MatMul ?



Slab recursive
FFLAS-FFPACK

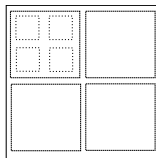


Tile recursive
FFLAS-FFPACK

- Sub-cubic complexity: recursive algorithms

The case of Gaussian elimination

Which reduction to MatMul ?

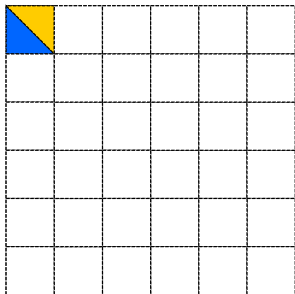


Tile recursive
FFLAS-FFPACK

- ▶ Sub-cubic complexity: recursive algorithms
- ▶ Data locality

Block algorithms

Tiled Iterative



Slab Recursive

Tiled Recursive

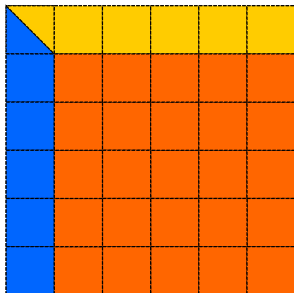
getrf: $A \rightarrow L, U$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive



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

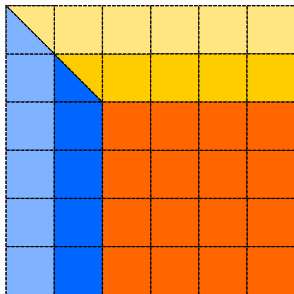
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive



getrf: $A \rightarrow L, U$

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

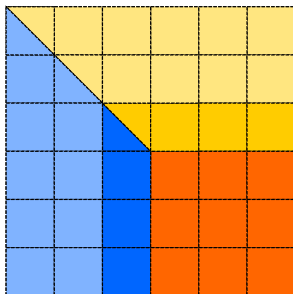
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative

Slab Recursive

Tiled Recursive



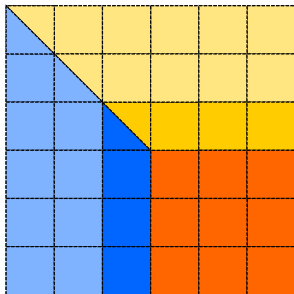
getrf: $A \rightarrow L, U$

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

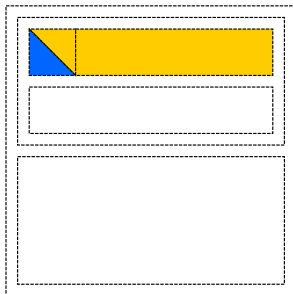
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative



Slab Recursive

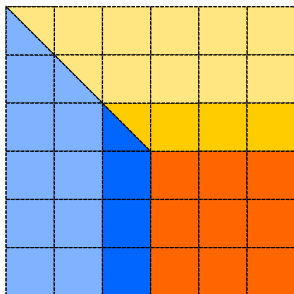


Tiled Recursive

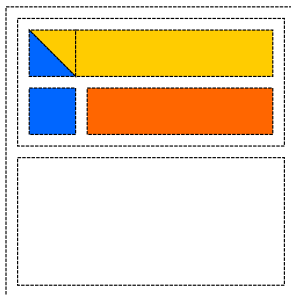
getrf: $A \rightarrow L, U$

Block algorithms

Tiled Iterative



Slab Recursive



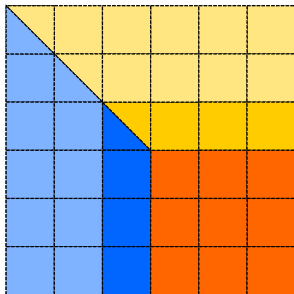
Tiled Recursive

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

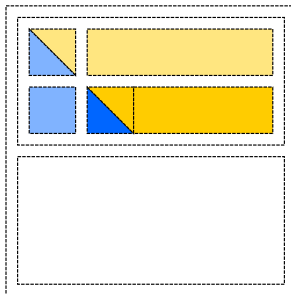
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative



Slab Recursive

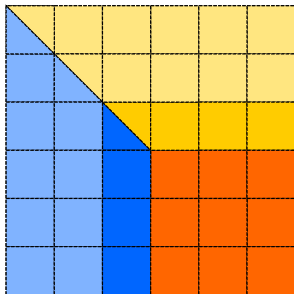


Tiled Recursive

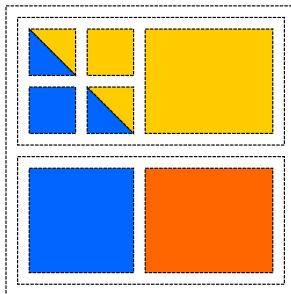
getrf: $A \rightarrow L, U$

Block algorithms

Tiled Iterative



Slab Recursive



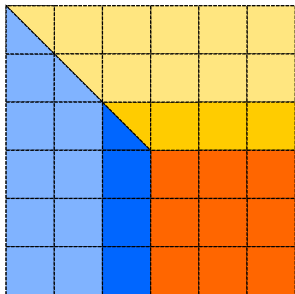
Tiled Recursive

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

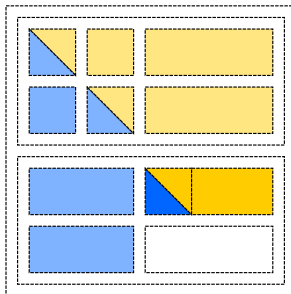
$$\text{gemm: } C \leftarrow C - A \times B$$

Block algorithms

Tiled Iterative



Slab Recursive

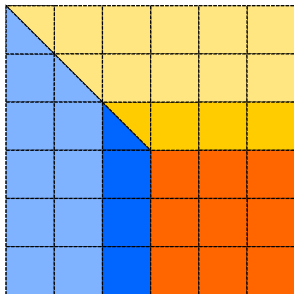


Tiled Recursive

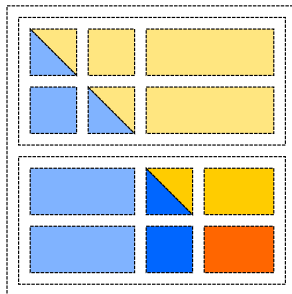
getrf: $A \rightarrow L, U$

Block algorithms

Tiled Iterative



Slab Recursive



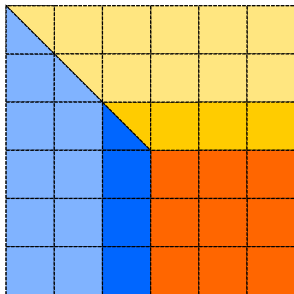
Tiled Recursive

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

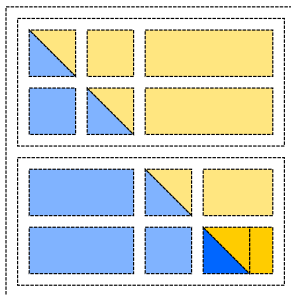
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tiled Iterative



Slab Recursive

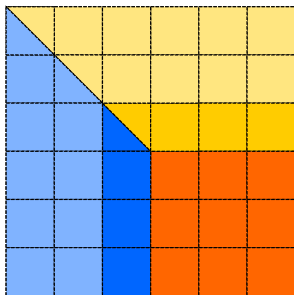


Tiled Recursive

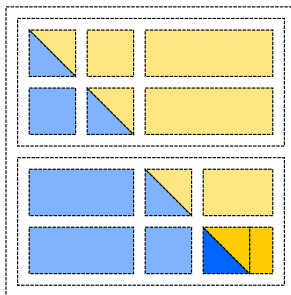
getrf: $A \rightarrow L, U$

Block algorithms

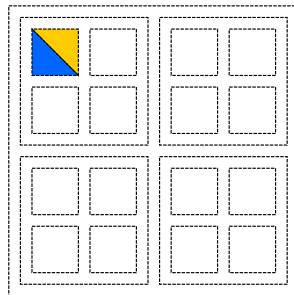
Tiled Iterative



Slab Recursive



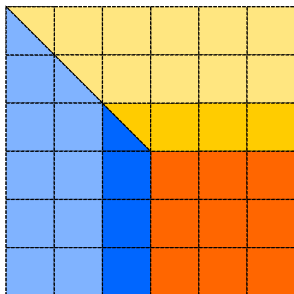
Tiled Recursive



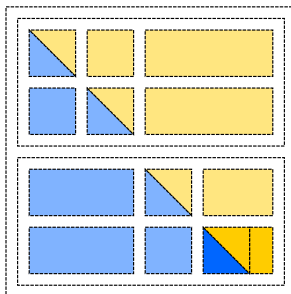
getrf: $A \rightarrow L, U$

Block algorithms

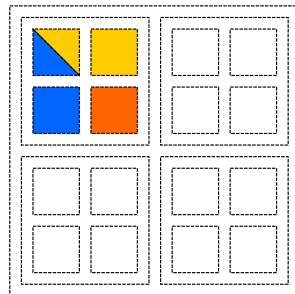
Tiled Iterative



Slab Recursive



Tiled Recursive

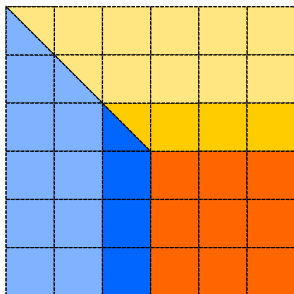


trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$

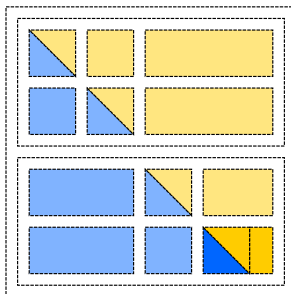
gemm: $C \leftarrow C - A \times B$

Block algorithms

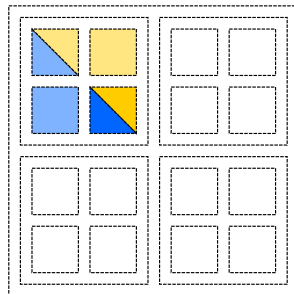
Tiled Iterative



Slab Recursive



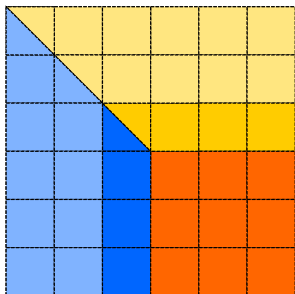
Tiled Recursive



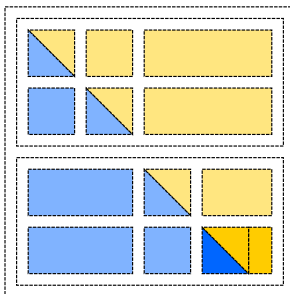
getrf: $A \rightarrow L, U$

Block algorithms

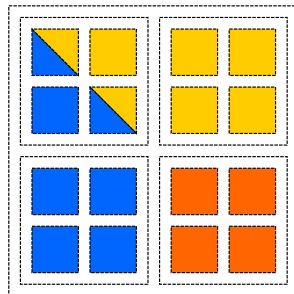
Tiled Iterative



Slab Recursive



Tiled Recursive

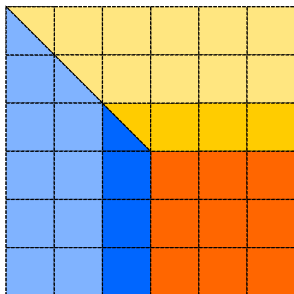


trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$

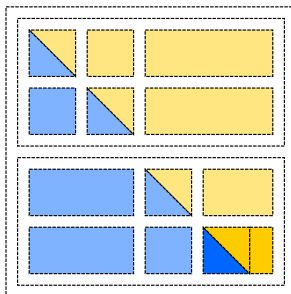
gemm: $C \leftarrow C - A \times B$

Block algorithms

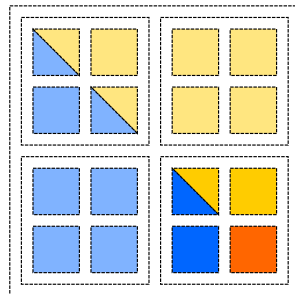
Tiled Iterative



Slab Recursive



Tiled Recursive



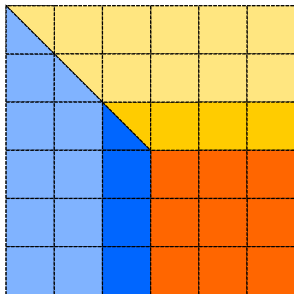
getrf: $A \rightarrow L, U$

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

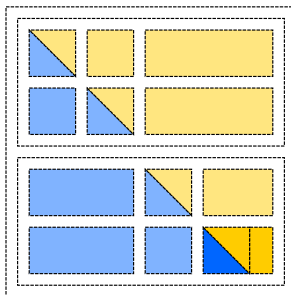
gemm: $C \leftarrow C - A \times B$

Block algorithms

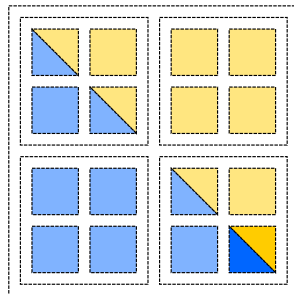
Tiled Iterative



Slab Recursive



Tiled Recursive



getrf: $A \rightarrow L, U$

Counting Modular Reductions

1	Tiled Iter. Right looking	$\frac{1}{3k} \mathbf{n}^3 + \left(1 - \frac{1}{k}\right) n^2 + \left(\frac{1}{6}k - \frac{5}{2} + \frac{3}{k}\right) n$
$\wedge$	Tiled Iter. Left looking	$\left(2 - \frac{1}{2k}\right) \mathbf{n}^2 + \left(-\frac{5}{2}k - 1 + \frac{2}{k}\right) n + 2k^2 - 2k + 1$
k^2	Tiled Iter. Crout	$\left(\frac{5}{2} - \frac{1}{k}\right) \mathbf{n}^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right) n + k^2$

Counting Modular Reductions

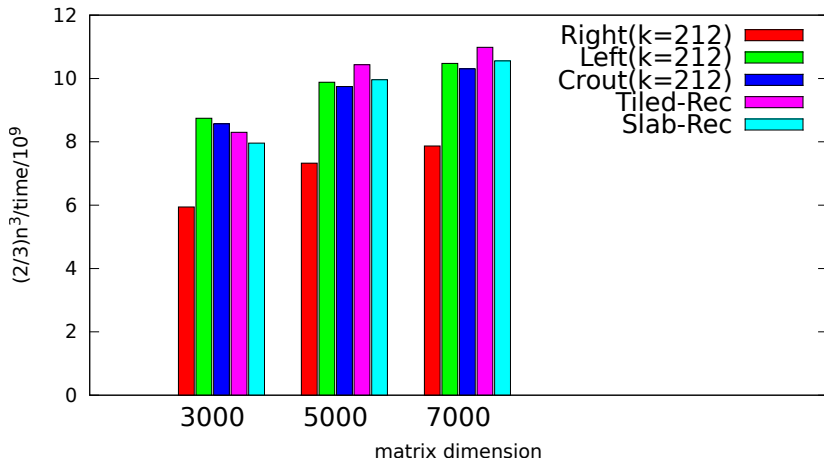
$k \geq 1$	Tiled Iter. Right looking	$\frac{1}{3k} \mathbf{n}^3 + \left(1 - \frac{1}{k}\right) n^2 + \left(\frac{1}{6}k - \frac{5}{2} + \frac{3}{k}\right) n$
	Tiled Iter. Left looking	$\left(2 - \frac{1}{2k}\right) \mathbf{n}^2 + \left(-\frac{5}{2}k - 1 + \frac{2}{k}\right) n + 2k^2 - 2k + 1$
	Tiled Iter. Crout	$\left(\frac{5}{2} - \frac{1}{k}\right) \mathbf{n}^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right) n + k^2$
$k = 1$	Iter. Right looking	$\frac{1}{3} \mathbf{n}^3 - \frac{1}{3} n$
	Iter. Left Looking	$\frac{3}{2} \mathbf{n}^2 - \frac{3}{2} n + 1$
	Iter. Crout	$\frac{3}{2} \mathbf{n}^2 - \frac{7}{2} n + 3$

Counting Modular Reductions

$k \geq 1$	Tiled Iter. Right looking	$\frac{1}{3k} \mathbf{n}^3 + \left(1 - \frac{1}{k}\right) n^2 + \left(\frac{1}{6}k - \frac{5}{2} + \frac{3}{k}\right) n$
	Tiled Iter. Left looking	$\left(2 - \frac{1}{2k}\right) \mathbf{n}^2 + \left(-\frac{5}{2}k - 1 + \frac{2}{k}\right) n + 2k^2 - 2k + 1$
	Tiled Iter. Crout	$\left(\frac{5}{2} - \frac{1}{k}\right) \mathbf{n}^2 + \left(-2k - \frac{5}{2} + \frac{3}{k}\right) n + k^2$
$k = 1$	Iter. Right looking	$\frac{1}{3} \mathbf{n}^3 - \frac{1}{3} n$
	Iter. Left Looking	$\frac{3}{2} \mathbf{n}^2 - \frac{3}{2} n + 1$
	Iter. Crout	$\frac{3}{2} \mathbf{n}^2 - \frac{7}{2} n + 3$
	Tiled Recursive	$2\mathbf{n}^2 - n \log_2 n - n$
	Slab Recursive	$\left(1 + \frac{1}{4} \log_2 \mathbf{n}\right) \mathbf{n}^2 - \frac{1}{2} n \log_2 n - n$

Impact in practice

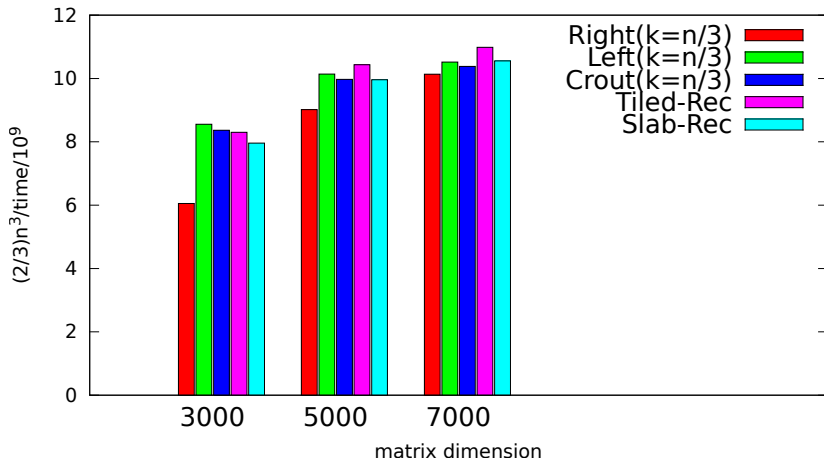
sequential LU decomposition variants on one core



As anticipated : Right-looking < Crout < Left-looking

Impact in practice

sequential LU decomposition variants on one core



As anticipated : Right-looking < Crout < Left-looking

Dealing with rank deficiencies and computing rank profiles

Rank profiles: first linearly independent columns

- ▶ Major invariant of a matrix (echelon form)
- ▶ Gröbner basis computations (Macaulay matrix)
- ▶ Krylov methods



Gaussian elimination revealing echelon forms:

[Ibarra, Moran and Hui 82]

$$A = L \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix} S \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix} P$$

[Keller-Gehrig 85]

$$X \quad A = R \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix}$$

[Jeannerod, P. and Storjohann 13]

$$A = P \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix} L \begin{matrix} \text{---} \\ \text{---} \\ \text{---} \end{matrix} E$$

Computing rank profiles

Lessons learned (or what we thought was necessary):

- ▶ treat rows in order
- ▶ exhaust all columns before considering the next row
- ▶ **slab** block splitting required (recursive or iterative)
 - ↪ similar to partial pivoting

Computing rank profiles

Lessons learned (or what we thought was necessary):

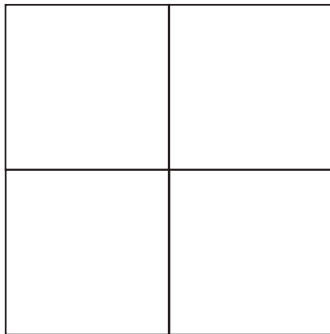
- ▶ treat rows in order
- ▶ exhaust all columns before considering the next row
- ▶ **slab** block splitting required (recursive or iterative)
 $\rightsquigarrow$ similar to partial pivoting

Tiled recursive PLUQ [Dumas P. Sultan 13,15]

- 1 Generalized to handle rank deficiency
 - ▶ 4 recursive calls necessary
 - ▶ in-place computation
- 2 Pivoting strategies exist to recover rank profile and echelon forms

A tiled recursive algorithm

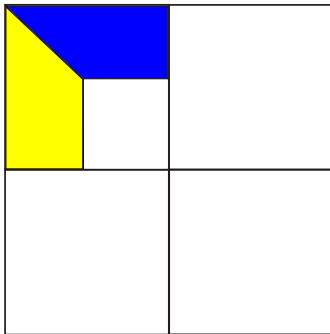
[Dumas, P. and Sultan 13]



2×2 block splitting

A tiled recursive algorithm

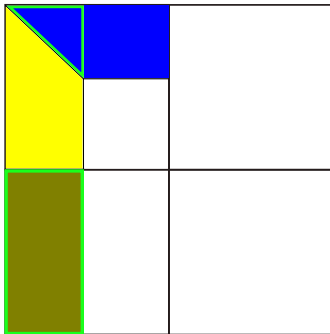
[Dumas, P. and Sultan 13]



Recursive call

A tiled recursive algorithm

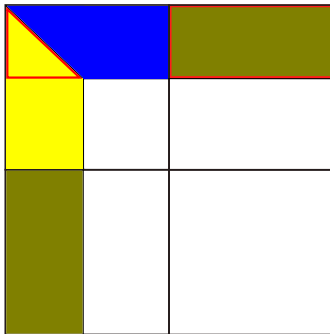
[Dumas, P. and Sultan 13]



TRSM: $B \leftarrow BU^{-1}$

A tiled recursive algorithm

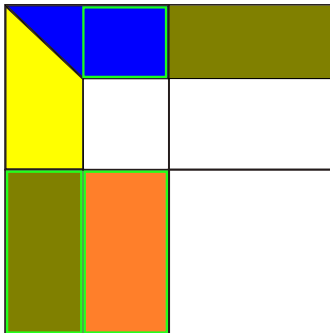
[Dumas, P. and Sultan 13]



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

A tiled recursive algorithm

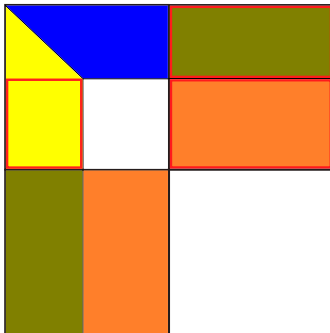
[Dumas, P. and Sultan 13]



MatMul: $C \leftarrow C - A \times B$

A tiled recursive algorithm

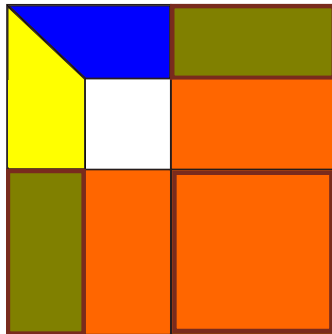
[Dumas, P. and Sultan 13]



MatMul: $C \leftarrow C - A \times B$

A tiled recursive algorithm

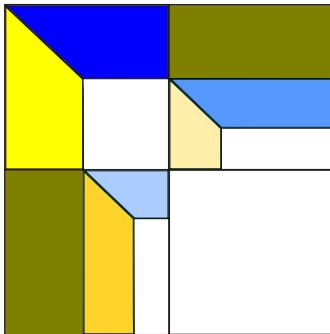
[Dumas, P. and Sultan 13]



MatMul: $C \leftarrow C - A \times B$

A tiled recursive algorithm

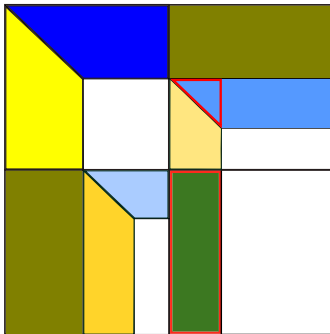
[Dumas, P. and Sultan 13]



2 independent recursive calls

A tiled recursive algorithm

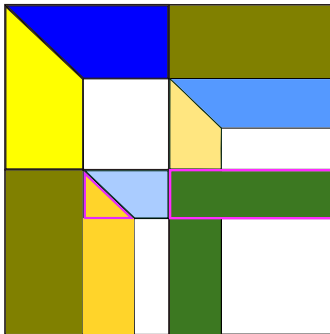
[Dumas, P. and Sultan 13]



TRSM: $B \leftarrow BU^{-1}$

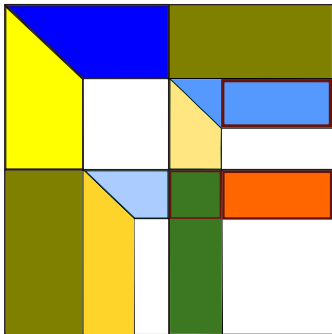
A tiled recursive algorithm

[Dumas, P. and Sultan 13]



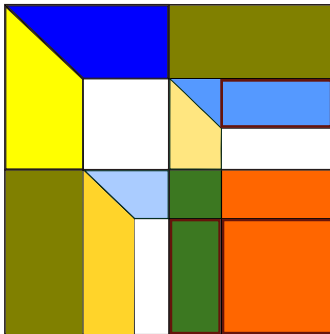
TRSM: $B \leftarrow L^{-1}B$

[Dumas, P. and Sultan 13]


$$\text{MatMul: } C \leftarrow C - A \times B$$

A tiled recursive algorithm

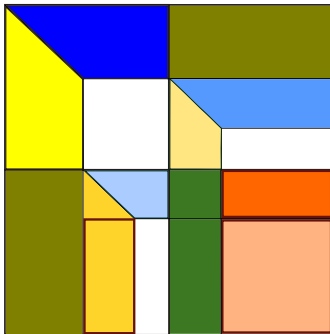
[Dumas, P. and Sultan 13]



MatMul: $C \leftarrow C - A \times B$

A tiled recursive algorithm

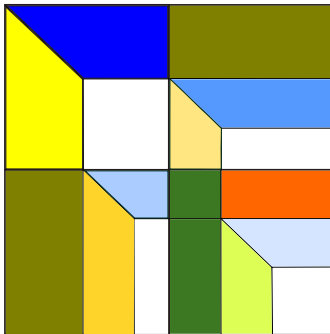
[Dumas, P. and Sultan 13]



MatMul: $C \leftarrow C - A \times B$

A tiled recursive algorithm

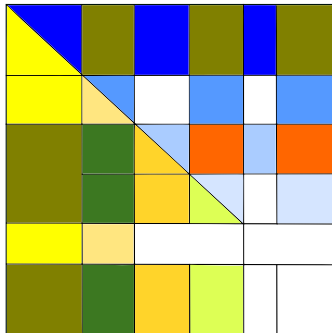
[Dumas, P. and Sultan 13]



Recursive call

A tiled recursive algorithm

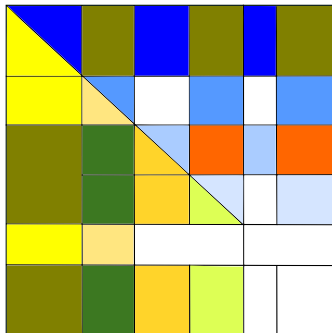
[Dumas, P. and Sultan 13]



Puzzle game (block cyclic rotations)

A tiled recursive algorithm

[Dumas, P. and Sultan 13]



- ▶ $O(mnr^{\omega-2})$ (degenerating to $2/3n^3$)
- ▶ computing col. and row rank profiles of all leading sub-matrices
- ▶ fewer modular reductions than slab algorithms
- ▶ rank deficiency introduces parallelism

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

A		$\mathcal{R}$
1 2 3 4	→	1 0 0 0
2 4 5 8		0 0 1 0
1 2 3 4		0 0 0 0
3 5 9 12		0 1 0 0

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A		$\mathcal{R}$
1 2 3 4	→	1 0 0 0
2 4 5 8		0 0 1 0
1 2 3 4		0 0 0 0
3 5 9 12		0 1 0 0

RowRP = {1}

ColRP = {1}

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A		$\mathcal{R}$
1 2 3 4	→	1 0 0 0
2 4 5 8		0 0 1 0
1 2 3 4		0 0 0 0
3 5 9 12		0 1 0 0

RowRP = {1,2}

ColRP = {1,3}

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A					$\mathcal{R}$			
1	2	3	4	→	1	0	0	0
2	4	5	8		0	0	1	0
1	2	3	4		0	0	0	0
3	5	9	12		0	1	0	0

RowRP = {1,4}

ColRP = {1,2}

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A		$\mathcal{R}$
1 2 3 4	→	1 0 0 0
2 4 5 8		0 0 1 0
1 2 3 4		0 0 0 0
3 5 9 12		0 1 0 0

RowRP = {1,4}

ColRP = {1,2}

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} \begin{bmatrix} U & V \\ & I_{n-r} \end{bmatrix} Q$$

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A					$\mathcal{R}$			
1	2	3	4	→	1	0	0	0
2	4	5	8		0	0	1	0
1	2	3	4		0	0	0	0
3	5	9	12		0	1	0	0

RowRP = {1,4}

ColRP = {1,2}

$$A = PLUQ = P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix} P^T P \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} Q Q^T \begin{bmatrix} U & V \\ & I_{n-r} \end{bmatrix} Q$$

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A					$\mathcal{R}$			
1	2	3	4	→	1	0	0	0
2	4	5	8		0	0	1	0
1	2	3	4		0	0	0	0
3	5	9	12		0	1	0	0

RowRP = {1,4}

ColRP = {1,2}

$$A = PLUQ = \underbrace{P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix}}_{\bar{L}} \underbrace{P^T P \begin{bmatrix} I_r & 0 \\ & 0 \end{bmatrix}}_{\Pi_{P,Q}} \underbrace{Q Q^T \begin{bmatrix} U & V \\ & I_{n-r} \end{bmatrix}}_{\bar{U}} Q$$

Computing all rank profiles at once



Dumas, P. and Sultan ISSAC'15 (Thursday 9 @ 3PM)

Definition (Rank Profile matrix)

The unique $\mathcal{R}_A \in \{0, 1\}^{m \times n}$ such that any pair of (i, j) -leading sub-matrix of $\mathcal{R}_A$ and of A have the same rank.

Theorem

- ▶ *RowRP and ColRP read directly on $\mathcal{R}(A)$*
- ▶ *Same holds for any (i, j) -leading submatrix.*

A		$\mathcal{R}$
1 2 3 4	→	1 0 0 0
2 4 5 8		0 0 1 0
1 2 3 4		0 0 0 0
3 5 9 12		0 1 0 0

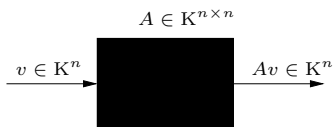
RowRP = {1,4}

ColRP = {1,2}

$$A = PLUQ = \underbrace{P \begin{bmatrix} L & 0 \\ M & I_{m-r} \end{bmatrix}}_{\bar{L}} \underbrace{P^T P \begin{bmatrix} I_r & 0 \\ & 0 \end{bmatrix}}_{\Pi_{P,Q}} \underbrace{Q Q^T \begin{bmatrix} U & V \\ & I_{n-r} \end{bmatrix}}_{\bar{U}} Q$$

With appropriate pivoting: $\Pi_{P,Q} = \mathcal{R}(A)$

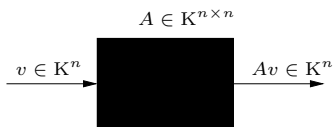
Reductions in black box linear algebra



Matrix-Vector Product: building block,
 $\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
 $\rightsquigarrow$ iterative Krylov/Lanczos methods
 $\rightsquigarrow O(nE(n) + n^2)$

Reductions in black box linear algebra

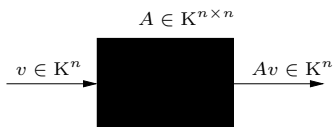


Matrix-Vector Product: building block,
 $\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
 $\rightsquigarrow$ iterative Krylov/Lanczos methods
 $\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen& Al. 02]
 $\rightsquigarrow$ reduces to MinPoly + preconditioners
 $\rightsquigarrow \tilde{O}(nE(n) + n^2)$

Reductions in black box linear algebra



Matrix-Vector Product: building block,
 $\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
 $\rightsquigarrow$ iterative Krylov/Lanczos methods
 $\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen& Al. 02]
 $\rightsquigarrow$ reduces to MinPoly + preconditioners
 $\rightsquigarrow \tilde{O}(nE(n) + n^2)$

Characteristic Poly.: [Dumas P. Saunders 09]
 $\rightsquigarrow$ reduces to MinPoly, Rank, ...

Reductions in black box linear algebra

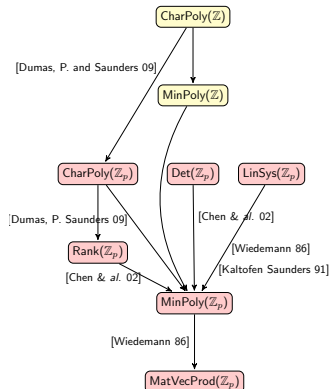


Matrix-Vector Product: building block,
 $\rightsquigarrow$ costs $E(n)$

Minimal polynomial: [Wiedemann 86]
 $\rightsquigarrow$ iterative Krylov/Lanczos methods
 $\rightsquigarrow O(nE(n) + n^2)$

Rank, Det, Solve: [Chen & Al. 02]
 $\rightsquigarrow$ reduces to MinPoly + preconditioners
 $\rightsquigarrow \tilde{O}(nE(n) + n^2)$

Characteristic Poly.: [Dumas P. Saunders 09]
 $\rightsquigarrow$ reduces to MinPoly, Rank, ...



Outline

- 1 Choosing the underlying arithmetic
 - Using boolean arithmetic
 - Using machine word arithmetic
 - Larger field sizes
- 2 Reductions and building blocks
 - In dense linear algebra
 - In blackbox linear algebra
- 3 Size dimension trade-offs
 - Hermite normal form
 - Frobenius normal form
- 4 Parallel exact linear algebra
 - Ingredients for the parallelization
 - Parallel dense linear algebra mod p

Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \dots$

Multimodular methods

over $K[X]$: evaluation-interpolation

over $\mathbb{Z}, \mathbb{Q}$: Chinese Remainder Theorem

$$\text{Cost} = \text{Algebraic Cost} \times \text{Size}(\text{Output})$$

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \dots$

Multimodular methods

over $K[X]$: evaluation-interpolation

over $\mathbb{Z}, \mathbb{Q}$: Chinese Remainder Theorem

$$\text{Cost} = \text{Algebraic Cost} \times \text{Size}(\text{Output})$$

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

Example

Hadamard's bound: $|\det(A)| \leq (\|A\|_{\infty} \sqrt{n})^n$.

$\text{LinSys}_{\mathbb{Z}}(n) = O(n^{\omega} \times n(\log n + \log \|A\|_{\infty}))$

Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \dots$

Multimodular methods

over $K[X]$: evaluation-interpolation

over $\mathbb{Z}, \mathbb{Q}$: Chinese Remainder Theorem

$$\text{Cost} = \text{Algebraic Cost} \times \text{Size}(\text{Output})$$

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

Example

Hadamard's bound: $|\det(A)| \leq (\|A\|_\infty \sqrt{n})^n$.

$$\text{LinSys}_{\mathbb{Z}}(n) = O(n^\omega \times n(\log n + \log \|A\|_\infty)) = O(n^{\omega+1} \log \|A\|_\infty)$$

Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \dots$

Lifting techniques

p-adic lifting: [Moenck & Carter 79, Dixon 82]

- ▶ One computation over $\mathbb{Z}_p$
- ▶ Iterative lifting of the solution to $\mathbb{Z}, \mathbb{Q}$

Example

$$\text{LinSys}_{\mathbb{Z}}(n) = O(n^3 \log \|A\|_{\infty}^{1+\epsilon})$$

Size Dimension trade-offs

Computing with coefficients of varying size: $\mathbb{Z}, \mathbb{Q}, K[X], \dots$

Lifting techniques

p-adic lifting: [Moenck & Carter 79, Dixon 82]

- ▶ One computation over $\mathbb{Z}_p$
- ▶ Iterative lifting of the solution to $\mathbb{Z}, \mathbb{Q}$

High order lifting : [Storjohann 02,03]

- ▶ Fewer iteration steps
- ▶ larger dimension in the lifting

Example

$$\text{LinSys}_{\mathbb{Z}}(n) = O(n^{\omega} \log \|A\|_{\infty})$$

Improving time Complexities

Matrix multiplication: door to fast linear algebra

- ▶ over $\mathbb{Z}$: $O(n^\omega M(\log \|A\|)) = \tilde{O}(n^\omega \log \|A\|)$

Improving time Complexities

Matrix multiplication: door to fast linear algebra

- ▶ over $\mathbb{Z}$: $O(n^\omega M(\log \|A\|)) = \tilde{O}(n^\omega \log \|A\|)$

Equivalence over $\mathbb{Z}$ or $K[X]$: Hermite normal form

- ▶ [Kannan & Bachem 79]: $\in P$
- ▶ [Chou & Collins 82]: $\tilde{O}(n^6 \log \|A\|)$
- ▶ [Domich & Al. 87], [Illiopoulos 89]: $\tilde{O}(n^4 \log \|A\|)$
- ▶ [Micciancio & Warinschi01]: $\tilde{O}(n^5 \log \|A\|^2)$,
 $\tilde{O}(n^3 \log \|A\|)$ heur.
- ▶ [Storjohann & Labahn 96]: $\tilde{O}(n^{\omega+1} \log \|A\|)$
- ▶ [Gupta & Storjohann 11]: $\tilde{O}(n^3 \log \|A\|)$

Improving time Complexities

Matrix multiplication: door to fast linear algebra

- ▶ over $\mathbb{Z}$: $O(n^\omega M(\log \|A\|)) = \tilde{O}(n^\omega \log \|A\|)$

Equivalence over $\mathbb{Z}$ or $\mathbb{K}[X]$: Hermite normal form

- ▶ [Kannan & Bachem 79]: $\in P$
- ▶ [Chou & Collins 82]: $\tilde{O}(n^6 \log \|A\|)$
- ▶ [Domich & Al. 87], [Illiopoulos 89]: $\tilde{O}(n^4 \log \|A\|)$
- ▶ [Micciancio & Warinschi01]: $\tilde{O}(n^5 \log \|A\|^2)$,
 $\tilde{O}(n^3 \log \|A\|)$ heur.
- ▶ [Storjohann & Labahn 96]: $\tilde{O}(n^{\omega+1} \log \|A\|)$
- ▶ [Gupta & Storjohann 11]: $\tilde{O}(n^3 \log \|A\|)$

Similarity over a field: Frobenius normal form

- ▶ [Giesbrecht 93]: $\tilde{O}(n^\omega)$ probabilistic
- ▶ [Storjohann 00]: $\tilde{O}(n^\omega)$ deterministic
- ▶ [P. & Storjohann 07]: $\tilde{O}(n^\omega)$ probabilistic

Improving time Complexities

Matrix multiplication: door to fast linear algebra

- ▶ over $\mathbb{Z}$: $O(n^\omega M(\log \|A\|)) = \tilde{O}(n^\omega \log \|A\|)$

Equivalence over $\mathbb{Z}$ or $K[X]$: Hermite normal form

- ▶ [Kannan & Bachem 79]: $\in P$
- ▶ [Chou & Collins 82]: $\tilde{O}(n^6 \log \|A\|)$
- ▶ [Domich & Al. 87], [Illiopoulos 89]: $\tilde{O}(n^4 \log \|A\|)$
- ▶ [Micciancio & Warinschi01]: $\tilde{O}(n^5 \log \|A\|^2)$,
 $\tilde{O}(n^3 \log \|A\|)$ heur.
- ▶ [Storjohann & Labahn 96]: $\tilde{O}(n^{\omega+1} \log \|A\|)$
- ▶ [Gupta & Storjohann 11]: $\tilde{O}(n^3 \log \|A\|)$

Similarity over a field: Frobenius normal form

- ▶ [Giesbrecht 93]: $\tilde{O}(n^\omega)$ probabilistic
- ▶ [Storjohann 00]: $\tilde{O}(n^\omega)$ deterministic
- ▶ [P. & Storjohann 07]: $\tilde{O}(n^\omega)$ probabilistic

Building blocks and reductions

In brief

Reductions to a building block

Matrix Mult: block rec. $\sum_{i=1}^{\log n} n \left(\frac{n}{2^i}\right)^{\omega-1} = O(n^\omega)$ (Gauss, REF)

Matrix Mult: Iterative $\sum_{k=1}^n k \left(\frac{n}{k}\right)^\omega = O(n^\omega)$ (Frobenius)

Linear Sys: over $\mathbb{Z}$ (Hermite Normal Form)

Size/dimension compromises

- ▶ Hermite normal form : rank 1 updates reducing the determinant
- ▶ Frobenius normal form : degree k , dimension n/k for $k = 1 \dots n$

Hermite normal form: naive algorithm

Reduced Echelon form over a ring:
$$\begin{bmatrix} p_1 & * & x_{1,2} & * & * & x_{1,3} & * \\ & & p_2 & * & * & x_{2,3} & * \\ & & & & & p_3 & * \end{bmatrix} \text{ with}$$

$$0 \leq x_{*,j} < p_j.$$

for $i = 1 \dots n$ **do**

$(g, t_i, \dots, t_n) = \text{xgcd}(A_{i,i}, A_{i+1,i}, \dots, A_{n,i})$

$L_i \leftarrow \sum_{j=i+1}^n t_j L_j$

for $j = i + 1 \dots n$ **do**

$L_j \leftarrow L_j - \frac{A_{j,i}}{g} L_i$

▷ eliminate

end for

for $j = 1 \dots i - 1$ **do**

$L_j \leftarrow L_j - \lfloor \frac{A_{j,i}}{g} \rfloor L_i$

▷ reduce

end for

end for

Computing modulo the determinant [Domich & Al. 87]

Property

- For A non-singular: $\max_i \sum_j H_{ij} \leq \det H = \det A$

Example

$$A = \begin{bmatrix} -5 & 8 & -3 & -9 & 5 & 5 \\ -2 & 8 & -2 & -2 & 8 & 5 \\ 7 & -5 & -8 & 4 & 3 & -4 \\ 1 & -1 & 6 & 0 & 8 & -3 \end{bmatrix}, H = \begin{bmatrix} 1 & 0 & 3 & 237 & -299 & 90 \\ 0 & 1 & 1 & 103 & -130 & 40 \\ 0 & 0 & 4 & 352 & -450 & 135 \\ 0 & 0 & 0 & 486 & -627 & 188 \end{bmatrix}$$

$$\det A = 1944$$

Computing modulo the determinant [Domich & Al. 87]

Property

- ▶ For A non-singular: $\max_i \sum_j H_{ij} \leq \det H = \det A$
- ▶ Every computation can be done modulo $d = \det A$:

$$U' \begin{bmatrix} A & \\ dI_n & I_n \end{bmatrix} = \begin{bmatrix} H & \\ & I_n \end{bmatrix}$$

Example

$$A = \begin{bmatrix} -5 & 8 & -3 & -9 & 5 & 5 \\ -2 & 8 & -2 & -2 & 8 & 5 \\ 7 & -5 & -8 & 4 & 3 & -4 \\ 1 & -1 & 6 & 0 & 8 & -3 \end{bmatrix}, H = \begin{bmatrix} 1 & 0 & 3 & 237 & -299 & 90 \\ 0 & 1 & 1 & 103 & -130 & 40 \\ 0 & 0 & 4 & 352 & -450 & 135 \\ 0 & 0 & 0 & 486 & -627 & 188 \end{bmatrix}$$

$$\det A = 1944$$

$$\rightsquigarrow O(n^3) \times M(n(\log n + \log \|A\|)) = O(n^5 \log \|A\|^2)$$

Computing modulo the determinant

- ▶ Pessimistic estimate on the arithmetic size
- ▶ d large but most coefficients of H are small
- ▶ *On average* : only the last few columns are *large*

$\rightsquigarrow$ Compute H' close to H but with small determinant

Computing modulo the determinant

- ▶ Pessimistic estimate on the arithmetic size
- ▶ d large but most coefficients of H are small
- ▶ *On average* : only the last few columns are *large*

$\rightsquigarrow$ Compute H' close to H but with small determinant

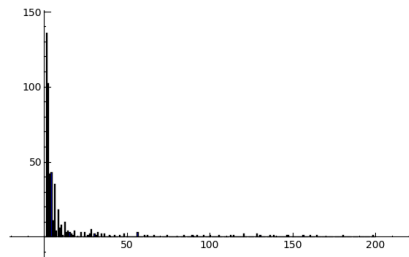
[Micciancio & Warinschi 01]

$$A = \begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$$

$$d_1 = \det \left(\begin{bmatrix} B \\ c^T \end{bmatrix} \right), d_2 = \det \left(\begin{bmatrix} B \\ d^T \end{bmatrix} \right)$$

$$g = \gcd(d_1, d_2) = sd_1 + td_2 \quad \text{Then}$$

$$\det \left(\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} \right) = g$$



Micciancio & Warinschi algorithm

Compute $d_1 = \det \left(\begin{bmatrix} B \\ c^T \end{bmatrix} \right), d_2 = \det \left(\begin{bmatrix} B \\ d^T \end{bmatrix} \right)$ ▷ Double Det

$(g, s, t) = \text{xgcd}(d_1, d_2)$

Compute H_1 the HNF of $\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} \bmod g$ ▷ Modular HNF

Recover H_2 the HNF of $\begin{bmatrix} B & b \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \end{bmatrix}$ ▷ AddCol

Recover H_3 the HNF of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$ ▷ AddRow

Micciancio & Warinschi algorithm

Compute $d_1 = \det \left(\begin{bmatrix} B \\ c^T \end{bmatrix} \right), d_2 = \det \left(\begin{bmatrix} B \\ d^T \end{bmatrix} \right)$

▷ Double Det

$(g, s, t) = \text{xgcd}(d_1, d_2)$

Compute H_1 the HNF of $\begin{bmatrix} B \\ sc^T + td^T \end{bmatrix} \bmod g$

▷ Modular HNF

Recover H_2 the HNF of $\begin{bmatrix} B & b \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \end{bmatrix}$

▷ AddCol

Recover H_3 the HNF of $\begin{bmatrix} B & b \\ c^T & a_{n-1,n} \\ d^T & a_{n,n} \end{bmatrix}$

▷ AddRow

Double Determinant

First approach: LU mod $p_1, \dots, p_k$ + CRT

- ▶ Only one elimination for the $n - 2$ first rows
- ▶ 2 updates for the last rows (triangular back substitution)
- ▶ k large such that $\prod_{i=1}^k p_i > n^n \log \|A\|^{n/2}$

Double Determinant

First approach: LU mod $p_1, \dots, p_k + \text{CRT}$

- ▶ Only one elimination for the $n - 2$ first rows
- ▶ 2 updates for the last rows (triangular back substitution)
- ▶ k *large* such that $\prod_{i=1}^k p_i > n^n \log \|A\|^{n/2}$

Second approach: [Abbott Bronstein Mulders 99]

- ▶ Solve $Ax = b$.
- ▶ $\delta = \text{lcm}(q_1, \dots, q_n)$ s.t. $x_i = p_i/q_i$

Then δ is a *large* divisor of $D = \det A$.

- ▶ Compute D/δ by LU mod $p_1, \dots, p_k + \text{CRT}$
- ▶ k *small*, such that $\prod_{i=1}^k p_i > n^n \log \|A\|^{n/2} / \delta$

Double Determinant : improved

Property

Let $x = [x_1, \dots, x_n]$ be the solution of $\left[\begin{array}{c|c} A & c \end{array} \right] x = d$. Then $y = \left[-\frac{x_1}{x_n}, \dots, -\frac{x_{n-1}}{x_n}, \frac{1}{x_n} \right]$ is the solution of $\left[\begin{array}{c|c} A & d \end{array} \right] y = c$.

- ▶ 1 system solve
- ▶ 1 LU for each p_i

$\rightsquigarrow d_1, d_2$ computed at about the cost of 1 déterminant

AddCol

Problem

Find a vector e such that

$$\left[H_1 \mid e \right] = U \begin{bmatrix} B & b \\ sc^T + td^T & sa_{n-1,n} + ta_{n,n} \end{bmatrix}$$

$$\begin{aligned} e &= U \begin{bmatrix} b \\ sa_{n-1,n} + ta_{n,n} \end{bmatrix} \\ &= H_1 \begin{bmatrix} B \\ sc^T + td^T \end{bmatrix}^{-1} \begin{bmatrix} b \\ sa_{n-1,n} + ta_{n,n} \end{bmatrix} \end{aligned}$$

$\rightsquigarrow$ Solve a system.

- ▶ $n - 1$ first rows are *small*
- ▶ last row is *large*

AddCol

Idea:

replace the last row by a random *small* one w^T .

$$\begin{bmatrix} B \\ w^T \end{bmatrix} y = \begin{bmatrix} b \\ a_{n-1,n-1} \end{bmatrix}$$

Let $\{k\}$ be a basis of the kernel of B . Then

$$x = y + \alpha k.$$

where

$$\alpha = \frac{a_{n-1,n-1} - (sc^T + td^T) \cdot y}{(sc^T + td^T) \cdot k}$$

$\rightsquigarrow$ limits the *expensive* arithmetic to a few dot products

Computing the Frobenius normal form

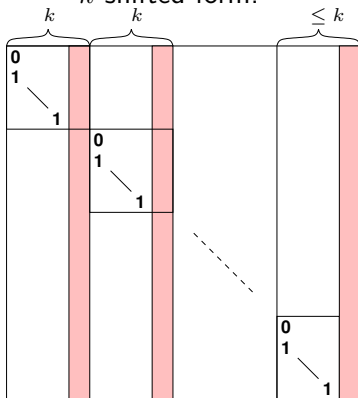
Definition

Unique $F = U^{-1}AU = \text{Diag}(C_{f_0}, \dots, C_{f_k})$ with $f_k | f_{k-1} | \dots | f_0$.

Computing the Frobenius normal form

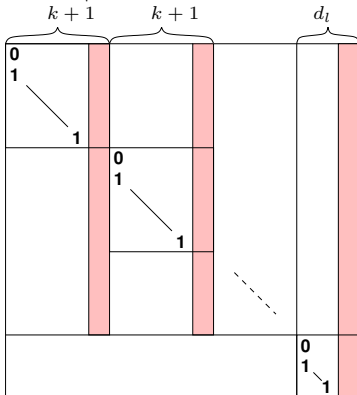
[P. & Storjohann 07]

k -shifted form:



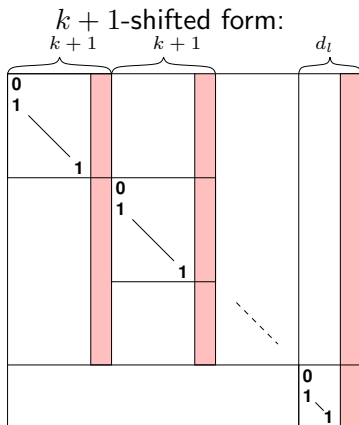
[P. & Storjohann 07]

$k + 1$ -shifted form:



Computing the Frobenius normal form

[P. & Storjohann 07]



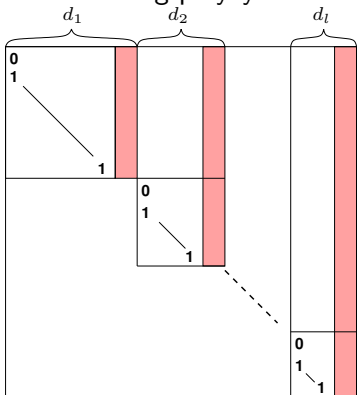
- ▶ From k to $k + 1$ -shifted in $O(k(\frac{n}{k})^\omega)$
- ▶ Compute iteratively from a 1-shifted form
- ▶ Invariant factors appear by increasing degree

Computing the Frobenius normal form

[P. & Storjohann 07]

Hessenberg polycyclic:

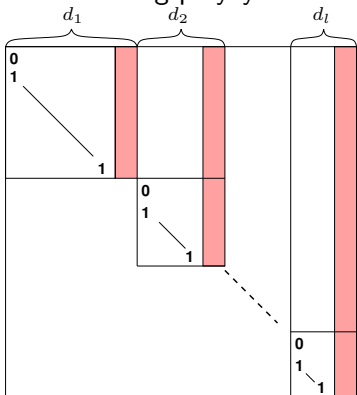
- ▶ From k to $k + 1$ -shifted in $O(k(\frac{n}{k})^\omega)$
- ▶ Compute iteratively from a 1-shifted form
- ▶ Invariant factors appear by increasing degree
- ▶ Until the Hessenberg polycyclic form



Computing the Frobenius normal form

[P. & Storjohann 07]

Hessenberg polycyclic:



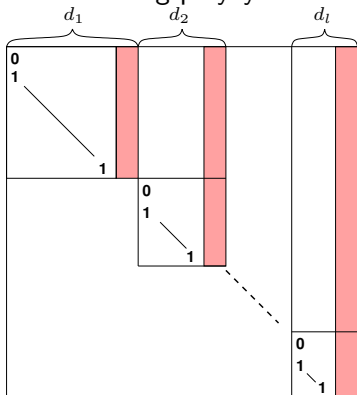
- ▶ From k to $k + 1$ -shifted in $O(k(\frac{n}{k})^\omega)$
- ▶ Compute iteratively from a 1-shifted form
- ▶ Invariant factors appear by increasing degree
- ▶ Until the Hessenberg polycyclic form

$$n^\omega \sum_{k=1}^n \left(\frac{1}{k}\right)^{\omega-1} \leq \zeta(\omega-1)n^\omega = O(n^\omega)$$

Computing the Frobenius normal form

[P. & Storjohann 07]

Hessenberg polycyclic:



- ▶ From k to $k + 1$ -shifted in $O(k(\frac{n}{k})^\omega)$
- ▶ Compute iteratively from a 1-shifted form
- ▶ Invariant factors appear by increasing degree
- ▶ Until the Hessenberg polycyclic form

$$n^\omega \sum_{k=1}^n \left(\frac{1}{k}\right)^{\omega-1} \leq \zeta(\omega-1)n^\omega = O(n^\omega)$$

- ▶ Generalized to the Frobenius form as well
- ▶ Transformation matrix in $O(n^\omega \log \log n)$

A new type size dimension trade-off

$$xI_n - A$$

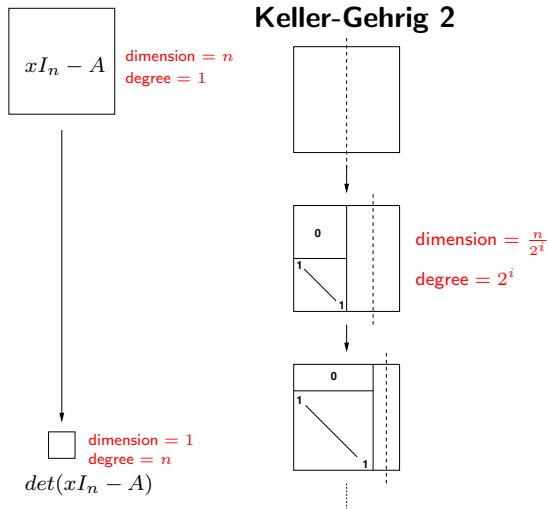
dimension = n
degree = 1



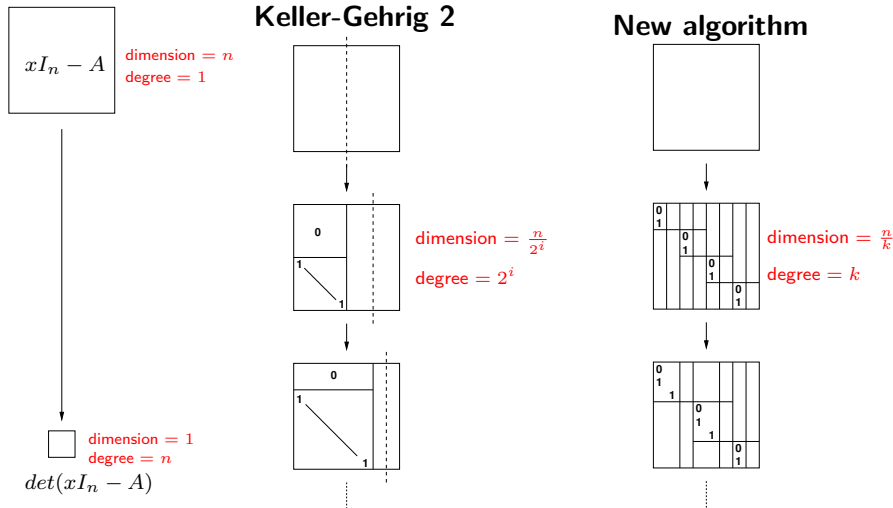
dimension = 1
degree = n

$$\det(xI_n - A)$$

A new type size dimension trade-off



A new type size dimension trade-off



Outline

- 1 Choosing the underlying arithmetic
 - Using boolean arithmetic
 - Using machine word arithmetic
 - Larger field sizes
- 2 Reductions and building blocks
 - In dense linear algebra
 - In blackbox linear algebra
- 3 Size dimension trade-offs
 - Hermite normal form
 - Frobenius normal form
- 4 Parallel exact linear algebra
 - Ingredients for the parallelization
 - Parallel dense linear algebra mod p

Parallelization

Parallel numerical linear algebra

- ▶ cost invariant wrt. splitting
 - ▷ $O(n^3)$
 - ↪ fine grain
 - ↪ block iterative algorithms
- ▶ regular task load
- ▶ Numerical stability constraints

Parallelization

Parallel numerical linear algebra

- ▶ cost invariant wrt. splitting
 - ▷ $O(n^3)$
 - ↪ fine grain
 - ↪ block iterative algorithms
- ▶ regular task load
- ▶ Numerical stability constraints

Exact linear algebra specificities

- ▶ cost affected by the splitting
 - ▷ $O(n^\omega)$ for $\omega < 3$
 - ▷ modular reductions
 - ↪ coarse grain
 - ↪ recursive algorithms
- ▶ rank deficiencies
 - ↪ unbalanced task loads

Ingredients for the parallelization

Criteria

- ▶ good performances
- ▶ portability across architectures
- ▶ abstraction for simplicity

Challenging key point: scheduling as a plugin

Program: only describes where the parallelism lies

Runtime: scheduling & mapping, depending on the context of execution

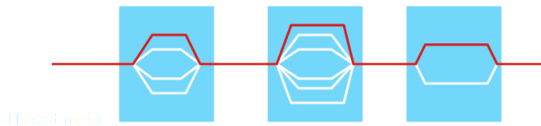
3 main models:

- ① Parallel loop [data parallelism]
- ② Fork-Join (independent tasks) [task parallelism]
- ③ Dependent tasks with data flow dependencies [task parallelism]

Data Parallelism

OMP

```
for (int step = 0; step < 2; ++step){  
#pragma omp parallel for  
    for (int i = 0; i < count; ++i)  
        A[i] = (B[i+1] + B[i-1] + 2.0*B[i])*0.25;  
}
```



Limitation: very un-efficient with recursive parallel regions

- ▶ Limited to iterative algorithms
- ▶ No composition of routines

Task parallelism with fork-Join

- ▶ Task based program: **spawn** + **sync**
- ▶ Especially suited for recursive programs

OMP (since v3)

```
void fibonacci(long* result, long n) {  
    if (n < 2)  
        *result = n;  
    else {  
        long x,y;  
#pragma omp task  
        fibonacci( &x, n-1 );  
        fibonacci( &y, n-2 );  
#pragma omp taskwait  
        *result = x + y;  
    }  
}
```

Tasks with dataflow dependencies

- ▶ Task based model avoiding synchronizations
- ▶ Infer synchronizations from the read/write specifications
 - ▷ A task is ready for execution when all its inputs variables are ready
 - ▷ A variable is ready when it has been written
- ▶ Recently supported: Athapascan [96], Kaapi [06], StarSs [07], StarPU [08], Quark [10], OMP since v4 [14]...

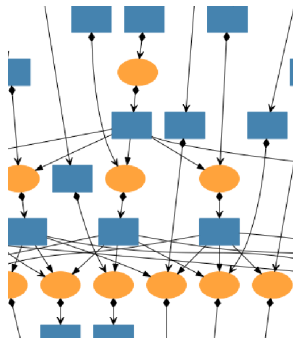
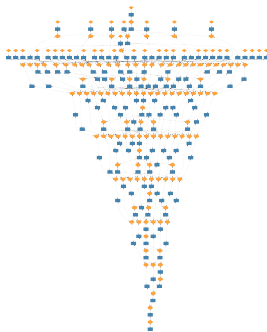


Illustration: Cholesky factorization

```

void Cholesky( double* A, int N, size_t NB ) {

    for ( size_t k=0; k < N; k += NB)
    {
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

        for ( size_t m=k+ NB; m < N; m += NB)
        {

            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                          NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }

        for ( size_t m=k+ NB; m < N; m += NB)
        {

            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                          NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

            for ( size_t n=k+NB; n < m; n += NB)
            {

                cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
                              NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
    }
}

```

Illustration: Cholesky factorization

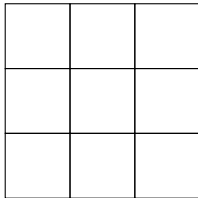
```

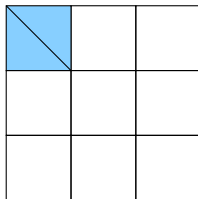
void Cholesky( double* A, int N, size_t NB ) {
#pragma omp parallel
#pragma omp single nowait
    for (size_t k=0; k < N; k += NB)
    {
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

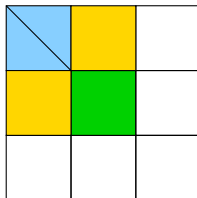
        for (size_t m=k+ NB; m < N; m += NB)
        {
#pragma omp task firstprivate(k, m) shared(A)
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }
#pragma omp taskwait // Barrier: no concurrency with next tasks
        for (size_t m=k+ NB; m < N; m += NB)
        {
#pragma omp task firstprivate(k, m) shared(A)
            cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

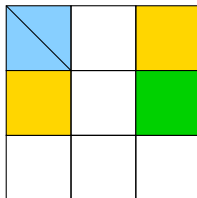
            for (size_t n=k+NB; n < m; n += NB)
            {
#pragma omp task firstprivate(k, m) shared(A)
                cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
                    NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
#pragma omp taskwait // Barrier: no concurrency with tasks at iteration k+1
    }
}

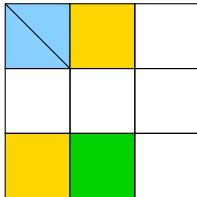
```

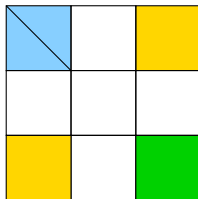













SYNC.

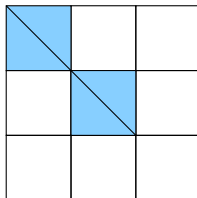


Illustration: Cholesky factorization

```

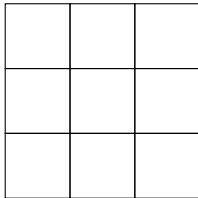
void Cholesky( double* A, int N, size_t NB ){
#pragma kaapi parallel
    for (size_t k=0; k < N; k += NB)
    {
#pragma kaapi task readwrite(&A[k*N+k]{ Id=N; [NB][NB]})
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

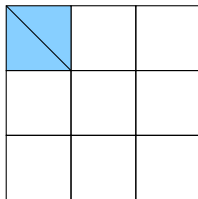
        for (size_t m=k+ NB; m < N; m += NB)
        {
#pragma kaapi task read(&A[k*N+k]{ Id=N; [NB][NB]}) readwrite(&A[m*N+k]{ Id=N; [NB][NB]})
            cbblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                          NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }

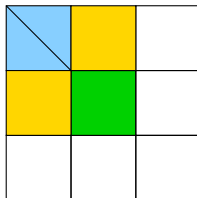
        for (size_t m=k+ NB; m < N; m += NB)
        {
#pragma kaapi task read(&A[m*N+k]{ Id=N; [NB][NB]}) readwrite(&A[m*N+m]{ Id=N; [NB][NB]})
            cbblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
                          NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

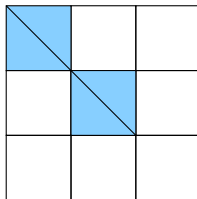
            for (size_t n=k+NB; n < m; n += NB)
            {
#pragma kaapi task read(&A[m*N+k]{ Id=N; [NB][NB]}, &A[n*N+k]{ Id=N; [NB][NB]})\
                      readwrite(&A[m*N+n]{ Id=N; [NB][NB]})
                cbblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
                              NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
            }
        }
    }
    // Implicit barrier only at the end of Kaapi parallel region
}

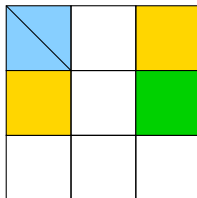
```

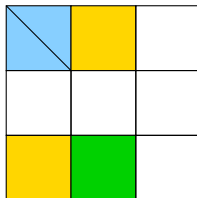


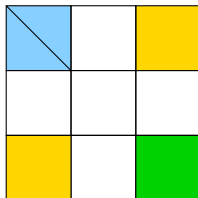






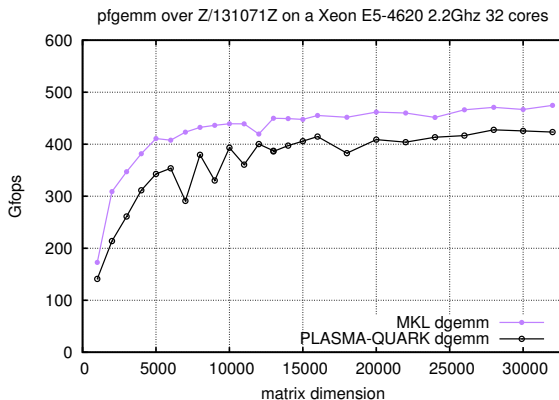
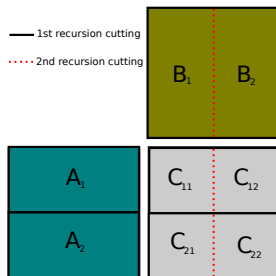






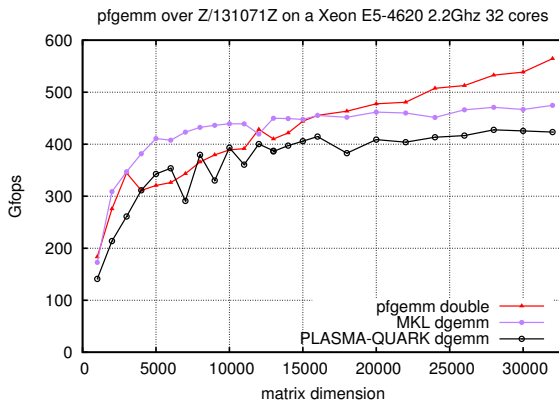
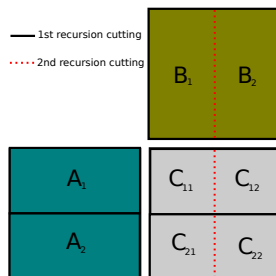
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]



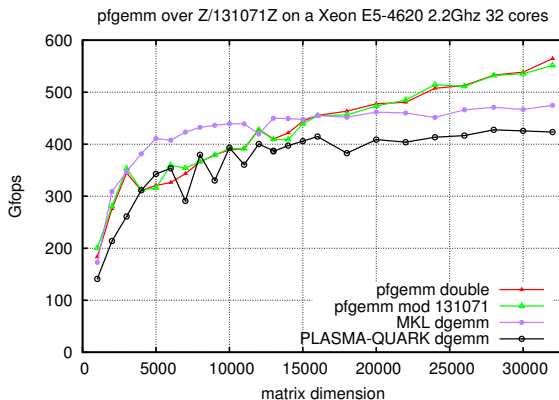
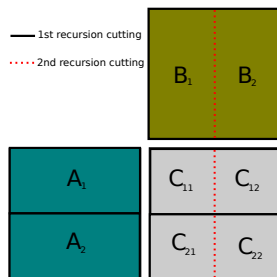
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]

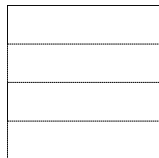


Parallel matrix multiplication

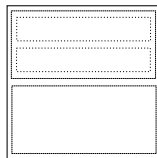
[Dumas, Gautier, P. & Sultan 14]



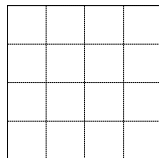
Gaussian elimination



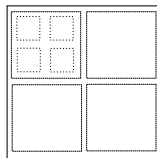
Slab iterative
LAPACK



Slab recursive
FFLAS-FFPACK

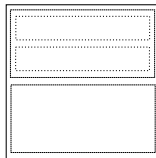


Tile iterative
PLASMA

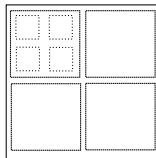


Tile recursive
FFLAS-FFPACK

Gaussian elimination



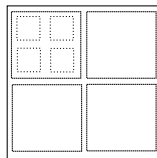
Slab recursive
FFLAS-FFPACK



Tile recursive
FFLAS-FFPACK

- Prefer recursive algorithms

Gaussian elimination

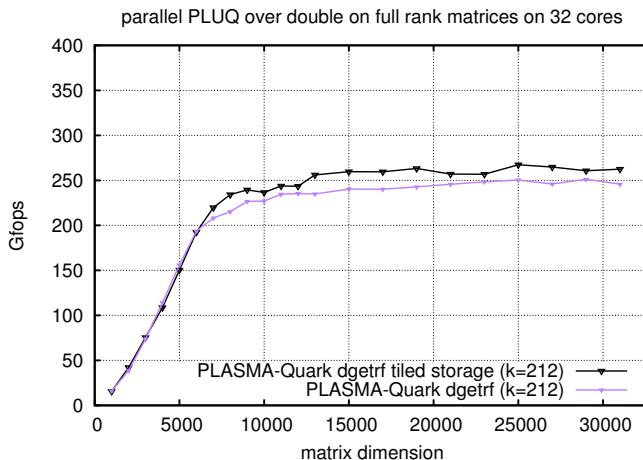


Tile recursive
FFLAS-FFPACK

- ▶ Prefer recursive algorithms
- ▶ Better data locality

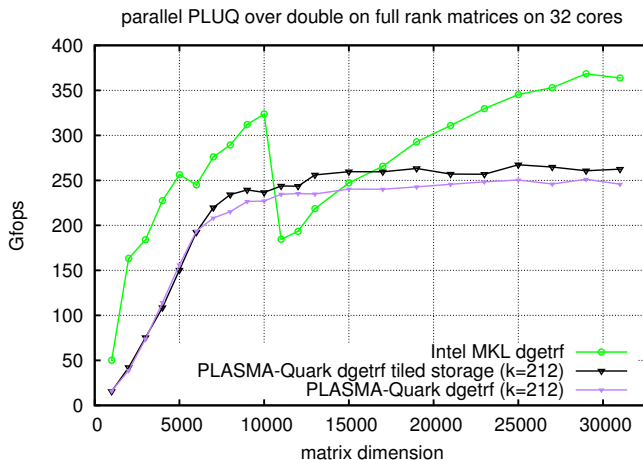
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)



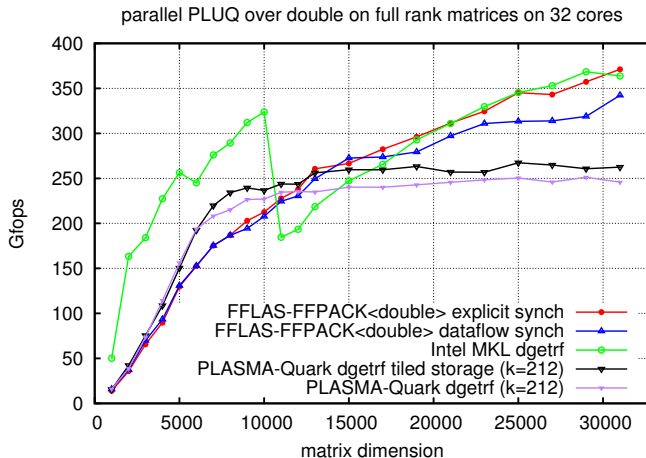
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)



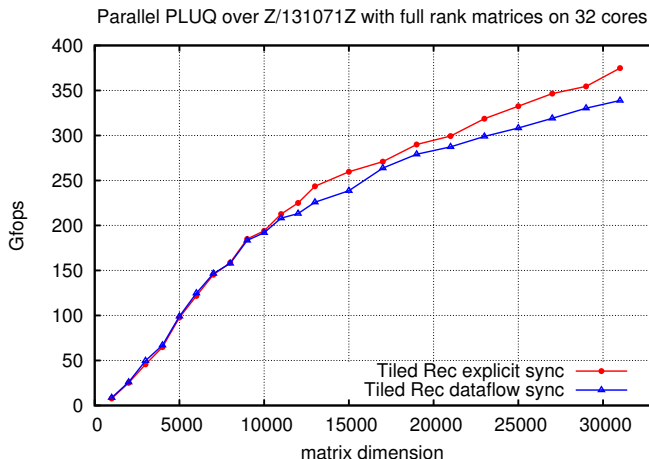
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Comparing numerical efficiency (no modulo)



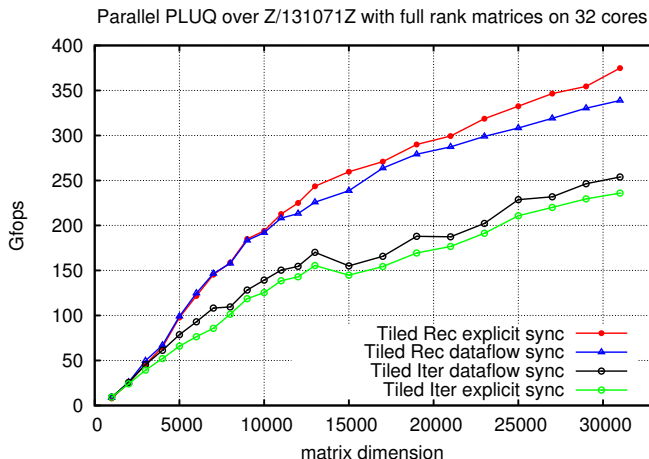
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over the finite field $\mathbb{Z}/131071\mathbb{Z}$



Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over the finite field $\mathbb{Z}/131071\mathbb{Z}$



Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction $>$ algorithm tuning $>$ building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed
- ▶ Medium size arithmetic: **RNS**
 - $\rightsquigarrow$ harnesses floating point efficiency
 - $\rightsquigarrow$ embarrassingly easy parallelization (and fault tolerance)

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed
- ▶ Medium size arithmetic: **RNS**
 - ↪ harnesses floating point efficiency
 - ↪ embarrassingly easy parallelization (and fault tolerance)
- ▶ Favor **tiled recursive** algorithms
 - ↪ **architecture oblivious vs aware** algorithms [Gustavson 07]

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed
- ▶ Medium size arithmetic: **RNS**
 - ↪ harnesses floating point efficiency
 - ↪ embarrassingly easy parallelization (and fault tolerance)
- ▶ Favor **tiled recursive** algorithms
 - ↪ **architecture oblivious vs aware** algorithms [Gustavson 07]
- ▶ New pivoting strategies revealing **all rank profile informations**
 - ↪ **tournament pivoting?** [Demmel, Grigori and Xiang 11]

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed
- ▶ Medium size arithmetic: **RNS**
 - ↪ harnesses floating point efficiency
 - ↪ embarrassingly easy parallelization (and fault tolerance)
- ▶ Favor **tiled recursive** algorithms
 - ↪ **architecture oblivious vs aware** algorithms [Gustavson 07]
- ▶ New pivoting strategies revealing **all rank profile informations**
 - ↪ **tournament pivoting?** [Demmel, Grigori and Xiang 11]
- ▶ Seek **size-dimension** trade-offs, even heuristic ones,

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction $>$ algorithm tuning $>$ building block implementation

- ▶ So far, **floating point** arithmetic delivers best speed
- ▶ Medium size arithmetic: **RNS**
 - ↪ harnesses floating point efficiency
 - ↪ embarrassingly easy parallelization (and fault tolerance)
- ▶ Favor **tiled recursive** algorithms
 - ↪ **architecture oblivious vs aware** algorithms [Gustavson 07]
- ▶ New pivoting strategies revealing **all rank profile informations**
 - ↪ **tournament pivoting**? [Demmel, Grigori and Xiang 11]
- ▶ Seek **size-dimension** trade-offs, even heuristic ones,
- ▶ **Recursive tasks** and **coarse grain** parallelization
 - ↪ Light weight task workstealing management required
 - ↪ Need for an improved recursive **dataflow** scheduling

Perspectives

Large scale distributed exact linear algebra

- ▶ reducing communications [Demmel, Grigori and Xiang 11]
- ▶ sparse and hybrid [Faugère and Lachartre 10]

Perspectives

Large scale distributed exact linear algebra

- ▶ reducing communications [Demmel, Grigori and Xiang 11]
- ▶ sparse and hybrid [Faugère and Lachartre 10]

Structured linear algebra

- ▶ A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- ▶ Combined with recent advances in linear algebra over $K[X]$
- ▶ Applications to list decoding

Perspectives

Large scale distributed exact linear algebra

- ▶ reducing communications [Demmel, Grigori and Xiang 11]
- ▶ sparse and hybrid [Faugère and Lachartre 10]

Structured linear algebra

- ▶ A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- ▶ Combined with recent advances in linear algebra over $K[X]$
- ▶ Applications to list decoding

Symbolic-numeric computation

- ▶ High precision floating point linear algebra via exact rational arithmetic and RNS

Perspectives

Large scale distributed exact linear algebra

- ▶ reducing communications [Demmel, Grigori and Xiang 11]
- ▶ sparse and hybrid [Faugère and Lachartre 10]

Structured linear algebra

- ▶ A lot of action recently [Jeannerod Schost 08], [Chowdhury & Al. 15]
- ▶ Combined with recent advances in linear algebra over $K[X]$
- ▶ Applications to list decoding

Symbolic-numeric computation

- ▶ High precision floating point linear algebra via exact rational arithmetic and RNS

Thank you