Computational Exact Linear Algebra

From Theory to Practice

Clément Pernet

Université Grenoble Alpes, LJK, France

OSCAR Summer School

September 9, 2021

Exact linear algebra

Matrices can be

- **Dense:** store all coefficients
- **Sparse:** store the non-zero coefficients only (and their location)
- **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 (and their location)

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 $\approx 100, 200, 1000, 2000, \ldots$ 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 (and their location)

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 $\approx 100, 200, 1000, 2000, \ldots$ 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 (and their location)
- 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 $\approx 100, 200, 1000, 2000, \ldots$ 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 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 GF$(q)[X]$, Structured

Need for high performance.

Content

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 algorihmic 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
   - A building block: matrix multiplication
   - Reductions to matrix multiplication
3. Size dimension trade-offs

Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Reductions and building blocks
   - A building block: matrix multiplication
   - Reductions to matrix multiplication
3. Size dimension trade-offs

Achieving high practical efficiency

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

$$\texttt{y} \leftarrow \texttt{y} \pm \texttt{a} * \texttt{b}$$

- 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
$\mathbb{Z}_p$	$>$ 32 bits
$\mathbb{Z}, \mathbb{Q}$	$>$ 26 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
$\mathbb{Z}_p$	$>$ 32 bits
$\mathbb{Z}, \mathbb{Q}$	$>$ 26 bits

Available CPU arithmetic units

- 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 f.p. arithmetic
$\mathbb{Z}_p$	> 32 bits	$\rightsquigarrow$ multiprec. ints, big ints, CRT
$\mathbb{Z}, \mathbb{Q}$	> 26 bits	$\rightsquigarrow$ multiprec. ints, big ints,CRT, lifting

Available CPU arithmetic units

- 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 f.p. arithmetic
$\mathbb{Z}_p$	$>$ 32 bits	$\rightsquigarrow$ multiprec. ints, big ints, CRT
$\mathbb{Z}, \mathbb{Q}$	$>$ 26 bits	$\rightsquigarrow$ multiprec. ints, big ints,CRT, lifting
$\mathsf{GF}(p^k) \equiv \mathbb{Z}_p[X]/(Q)$		$\rightsquigarrow$ Polynomial, Kronecker, Zech log, ...

Available CPU arithmetic units

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

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

$\texttt{uint64\_t} \equiv (\mathbb{Z}_2)^{64} \rightsquigarrow$

- ^ : bit-wise XOR, (+ mod 2)
- & : bit-wise AND, (× mod 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:

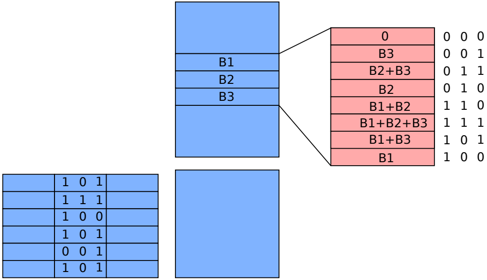
- avoid computing parities
- balance computation vs communication
- (slight) complexity improvement possible

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

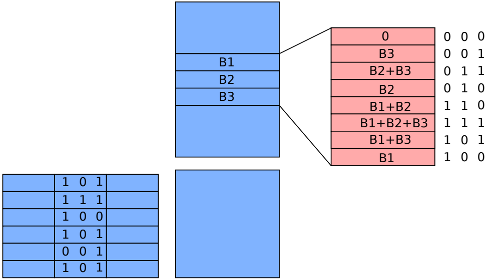


Tabulation:

- 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



- **For** $k = \log n \rightsquigarrow O(n^3/\log n)$**.**
- **In pratice: 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	111.82s
SIMD + 4 Russians	0.256s	2.829s	29.28s
SIMD + 4 Russians + Strassen	0.257s	2.001s	15.73s

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

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.

- either a priori:

Representation of $\mathbb{Z}_p$	$\{0 \ldots p-1\}$	$\{-\frac{p-1}{2} \ldots \frac{p-1}{2}\}$
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.

- either 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)	$\left(\frac{1+3^\ell}{2}\right)^2 \left\lfloor \frac{n}{2^\ell} \right\rfloor (p-1)^2$	$9^\ell \left\lfloor \frac{n}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2$

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.

- either 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)	$\left(\frac{1+3^\ell}{2}\right)^2 \left\lfloor \frac{n}{2^\ell} \right\rfloor (p-1)^2$	$9^\ell \left\lfloor \frac{n}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2$

- or 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 += a * b`: correct if $|ab + y| < 2^{63} \leadsto |a|, |b| < 2^{31.5}$

Computing over fixed size integers

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

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

   `y += a * b`: correct if $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31.5}$

```
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 $|ab + y| < 2^{63} \leadsto |a|, |b| < 2^{31.5}$

```
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 $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31.5}$

```
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 $|ab + y| < 2^{53} \rightsquigarrow |a|, |b| < 2^{26.5}$

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 $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31.5}$

```
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 $|ab + y| < 2^{53} \rightsquigarrow |a|, |b| < 2^{26.5}$

```
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 $|ab + y| < 2^{63} \leadsto |a|, |b| < 2^{31.5}$

```
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 $|ab + y| < 2^{53} \leadsto |a|, |b| < 2^{26.5}$

```
movsd   (%rax,%rdx,8), %xmm0
mulsd   -56(%rbp), %xmm0
addsd   %xmm0, %xmm1
movq    %xmm1, %rax
```

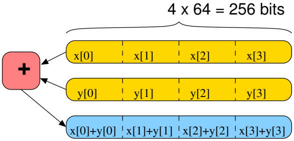
```
vinsertf128  $0x1, 16(%rcx,%rax), %ymm0,
vmulpd       %ymm1, %ymm0, %ymm0
vaddpd       (%rsi,%rax),%ymm0, %ymm0
vmovapd      %ymm0, (%rsi,%rax)
```

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

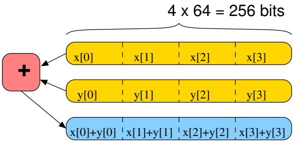


Exploiting *in-core* parallelism

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

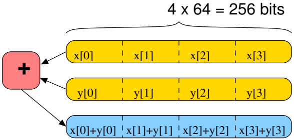
throughput of 1 op/ Cycle for all arithmetic ops considered here

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



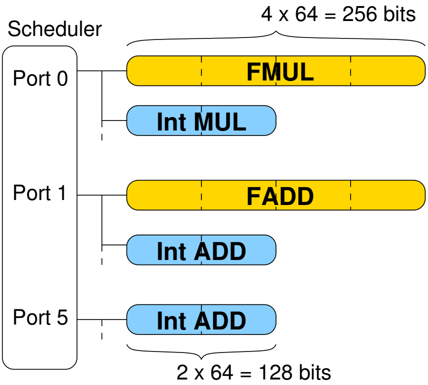
Pipeline: amortize the latency of an operation when used repeatedly

throughput of 1 op/ Cycle for all arithmetic ops considered here

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

SIMD and vectorization

Intel Sandybridge micro-architecture



Performs at every clock cycle:

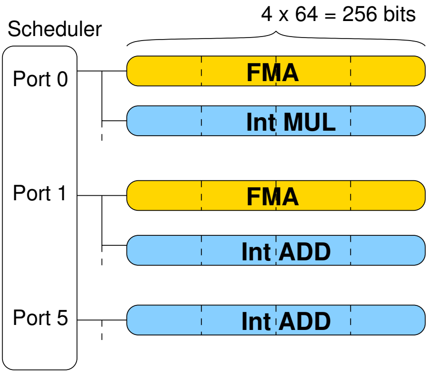
- 1 Floating Pt. Mul $\times$ 4
- 1 Floating Pt. Add $\times$ 4

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

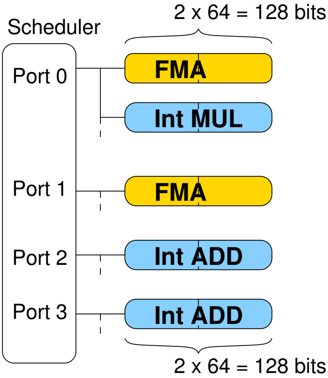
Or:

- 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

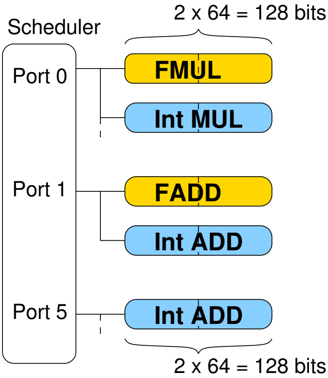
Or:

- 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	# axpy /Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Haswell	INT	256	2	1		4	3.5	**28**	
AVX2	FP	256			2	8	3.5	**56**	
Intel Sandybridge	INT								
AVX1	FP								
AMD Bulldozer	INT								
FMA4	FP								
Intel Nehalem	INT								
SSE4	FP								

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light $=$ CPU freq $\times$ ($\#$ axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ ($\#$ axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

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

Speed of light = CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# axpy /Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Skylake AVX512F	INT	512	2	1		8	3.7	**59**	
	FP	512			2	16	3.7	**118**	
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**

Speed of light $=$ CPU freq $\times$ (# axpy / Cycle) $\times 2$

Summary: 64 bits AXPY throughput

		Register size	# Adders	# Multipliers	# FMA	# axpy /Cycle	CPU Freq. (Ghz)	Speed of Light (Gfops)	Speed in practice (Gfops)
Intel Skylake	INT	512	2	1		8	3.7	**59**	
AVX512F	FP	512			2	16	3.7	**118**	**104.6**
Intel Haswell	INT	256	2	1		4	3.5	**28**	**23.3**
AVX2	FP	256			2	8	3.5	**56**	**49.2**
Intel Sandybridge	INT	128	2	1		2	3.3	**13.2**	**12.1**
AVX1	FP	256	1	1		4	3.3	**26.4**	**24.6**
AMD Bulldozer	INT	128	2	1		2	2.1	**8.4**	**6.44**
FMA4	FP	128			2	4	2.1	**16.8**	**13.1**
Intel Nehalem	INT	128	2	1		2	2.66	**10.6**	**4.47**
SSE4	FP	128	1	1		2	2.66	**10.6**	**9.6**

Speed of light $=$ CPU freq $\times$ (# axpy / Cycle) $\times 2$

Computing over fixed size integers: ressources

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

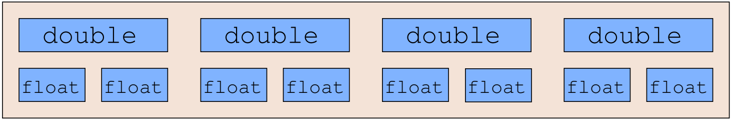
Experiments:

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

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

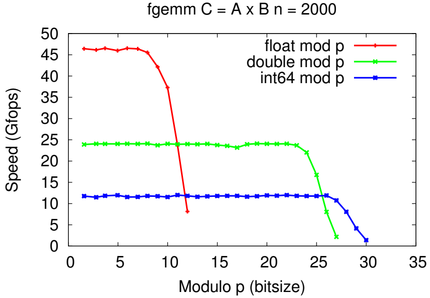
Integer Packing

32 bits: half the precision twice the speed



Gfops	double	float	int64_t	int32_t
Intel Skylake	104.6	202.3		
Intel Haswell	49.2	77.6	23.3	27.4
Intel SandyBridge	24.7	49.1	12.1	24.7
AMD Bulldozer	13.0	20.7	6.63	11.8

Computing over fixed size integers

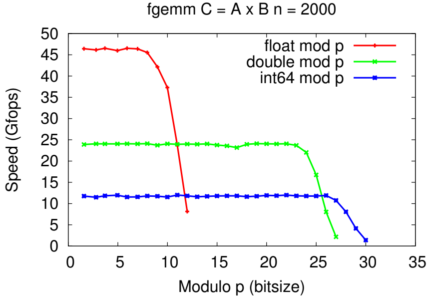


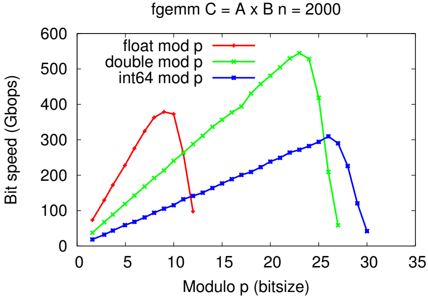
SandyBridge i5-3320M@3.3Ghz. $n = 2000$.

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

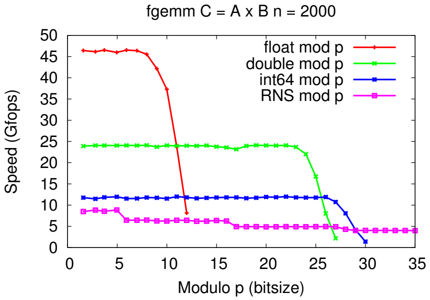
Which integer arithmetic?

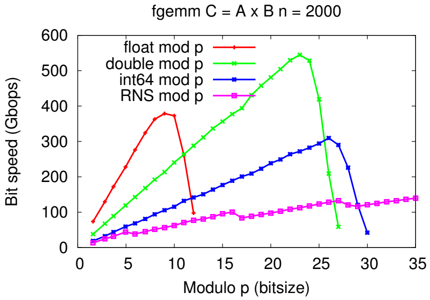
1. big integers (GMP)
2. fixed size multiprecision (Givaro-RecInt)
3. Residue Number Systems (Chinese Remainder theorem)
   $\rightsquigarrow$ using moduli delivering optimum bitspeed

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

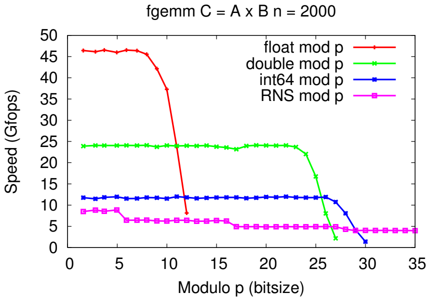
$n = 1000$, on an Intel SandyBridge.

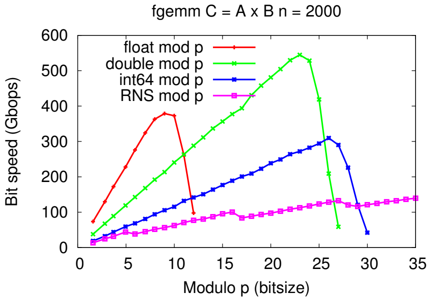
In practice

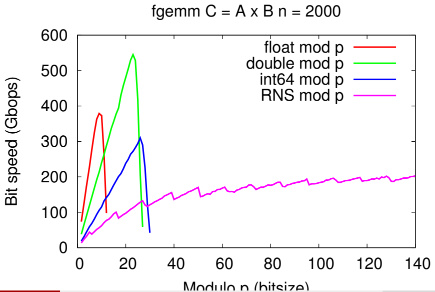

fgemm C = A x B n = 2000
float mod p
double mod p
int64 mod p
RNS mod p
Speed (Gfops)
Modulo p (bitsize)
0
5
10
15
20
25
30
35
40
45
50


fgemm C = A x B n = 2000
float mod p
double mod p
int64 mod p
RNS mod p
Bit speed (Gbops)
Modulo p (bitsize)
0
100
200
300
400
500
600

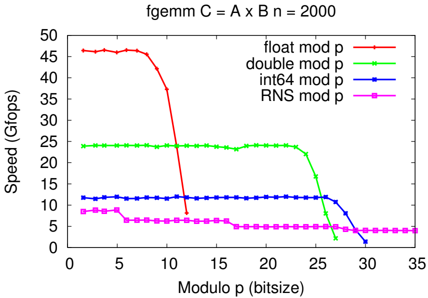
In practice

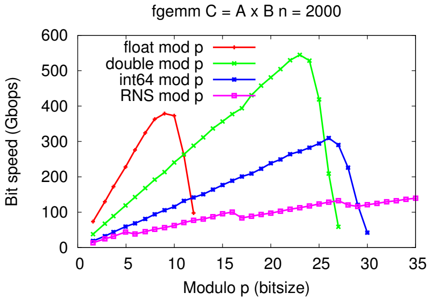

fgemm C = A x B n = 2000
Speed (Gfops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p

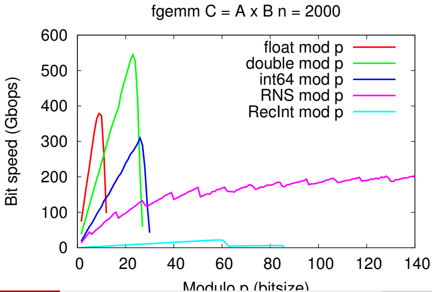

fgemm C = A x B n = 2000
Bit speed (Gbops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p


fgemm C = A x B n = 2000
Bit speed (Gbops)
float mod p
double mod p
int64 mod p
RNS mod p

In practice


fgemm C = A x B n = 2000
Speed (Gfops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p


fgemm C = A x B n = 2000
Bit speed (Gbops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p


fgemm C = A x B n = 2000
Bit speed (Gbops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p
RecInt mod p

Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Reductions and building blocks
   - A building block: matrix multiplication
   - Reductions to matrix multiplication
3. Size dimension trade-offs

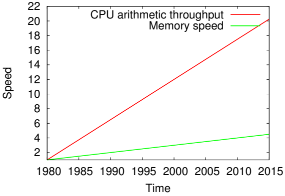
Reductions to building blocks

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

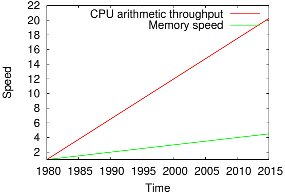
Memory access pattern

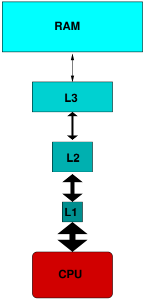
- **The memory wall:** communication speed improves slower than arithmetic



Memory access pattern

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





Memory access pattern

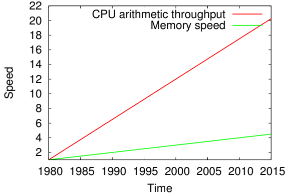
- **The memory wall:** communication speed improves slower than arithmetic
- Deep memory hierarchy

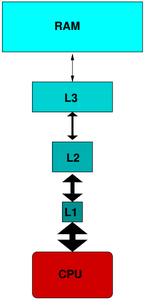
$\rightsquigarrow$ Need to overlap communications by computation

Design of BLAS 3 [Dongarra & Al. 87]

- Group all ops in Matrix products gemm:
  Work $O(n^3) >> $ Data $O(n^2)$

MatMul has become a building block in practice





Sub-cubic linear algebra

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

Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskĭi, 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]	$O(n^{2.3728639})$
[Alman, Vassilevska Wil. 20]	$O(n^{2.3728596})$

Sub-cubic linear algebra

< 1969: $O(n^3)$ for everyone (Gauss, Householder, Danilevskĭi, 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]	$O(n^{2.3728639})$
[Alman, Vassilevska Wil. 20]	$O(n^{2.3728596})$

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 $O(n^\omega)$
[P., Neiger 21]:	CharPoly in $O(n^\omega)$

Sub-cubic linear algebra

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

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

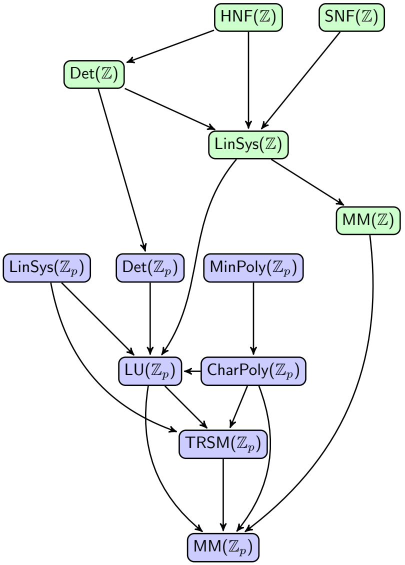
[Strassen 69]:	$O(n^{2.807})$
⋮	
[Schönhage 81]	$O(n^{2.52})$
⋮	
[Coppersmith, Winograd 90]	$O(n^{2.375})$
⋮	
[Le Gall 14]	$O(n^{2.3728639})$
[Alman, Vassilevska Wil. 20]	$O(n^{2.3728596})$

Other operations

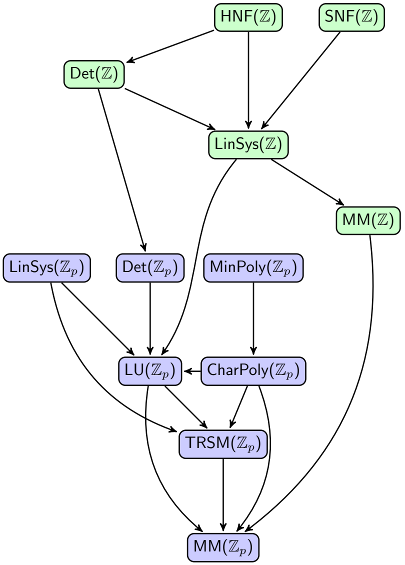
[Strassen 69]:	Inverse in $O(n^\omega)$
[Schönhage 72]:	QR in $O(n^\omega)$
[Bunch, Hopcroft 74]:	LU in $O(n^\omega)$
[Ibarra & *al.* 82]:	Rank in $O(n^\omega)$
[P., Neiger 21]:	CharPoly in $O(n^\omega)$

MatMul has become a building block in theoretical reductions

Reductions: theory



Reductions: theory

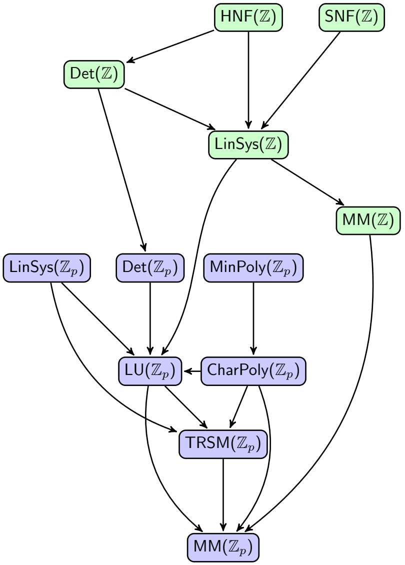


Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Reductions: theory and practice



Common mistrust

Fast linear algebra is

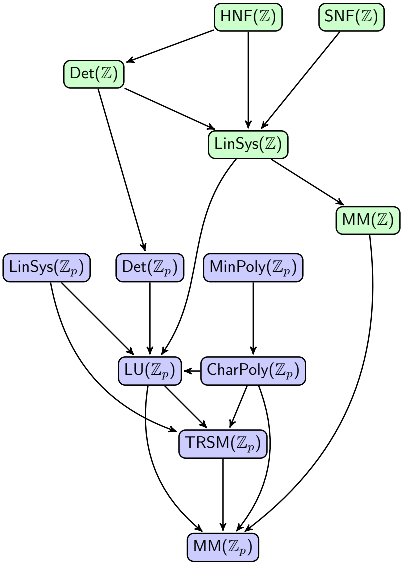
- ✗ never faster
- ✗ numerically unstable

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

Roadmap for efficiency in practice

1. Tune the MatMul building block.
2. Tune the reductions.
3. New reductions.

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

Ingedients [FFLAS-FFPACK library]

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

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

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

Ingedients [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

  $\rightsquigarrow$ numerical BLAS

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

Ingedients [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

  $\rightsquigarrow$ numerical BLAS
- Strassen-Winograd $6n^{2.807} + \dots$

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

Ingedients [FFLAS-FFPACK library]

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

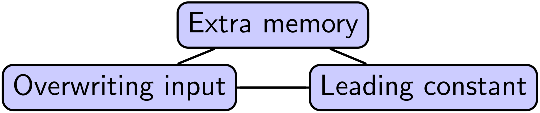
$\rightsquigarrow$ $9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2 < 2^{53}$

- Fastest integer arithmetic: `double`
- Cache optimizations

$\rightsquigarrow$ numerical BLAS

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

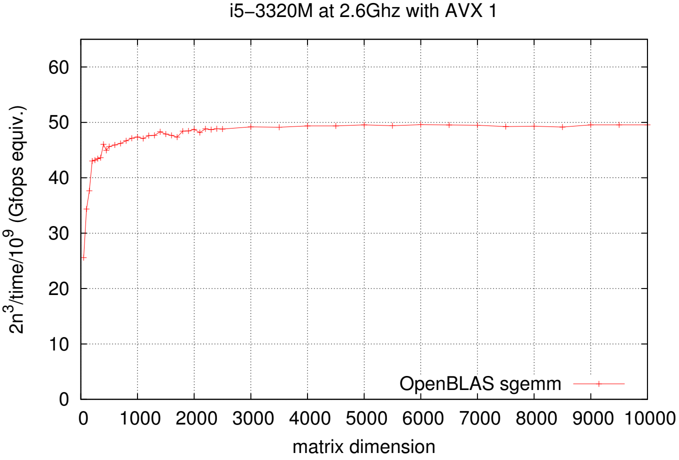
with memory efficient schedules [Boyer, Dumas, P. and Zhou 09]
Tradeoffs:



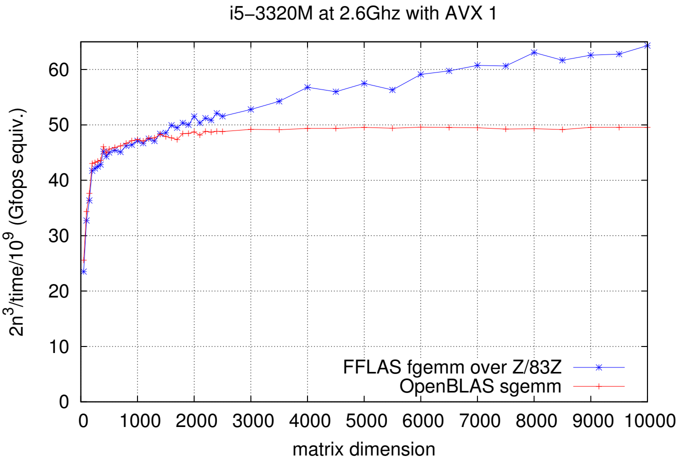
Fully in-place in

$7.2n^{2.807} + \ldots$

Sequential Matrix Multiplication

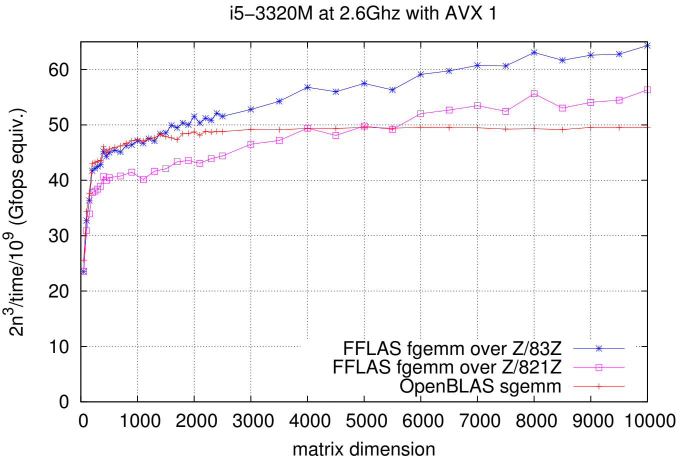


Sequential Matrix Multiplication



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

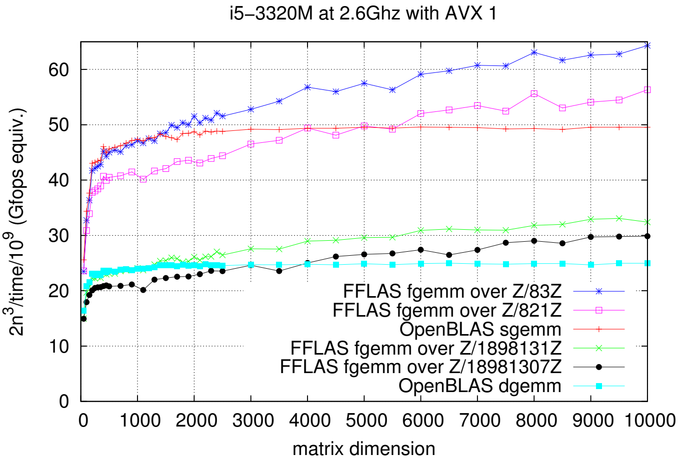
Sequential Matrix Multiplication



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

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

Sequential Matrix Multiplication



$p = 83$, $\leadsto$ 1 mod / 10000 mul. $\quad$ $p = 1898131$, $\leadsto$ 1 mod / 10000 mul.

$p = 821$, $\leadsto$ 1 mod / 100 mul. $\quad$ $p = 18981307$, $\leadsto$ 1 mod / 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 former probabilistic 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 former probabilistic 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**

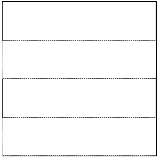
Intel Ivy-Bridge i5-3320 2.6Ghz using OpenBLAS-0.2.9

$\times 7.5$

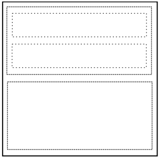
$\times 6.7$

The case of Gaussian elimination

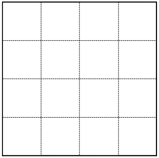
Which reduction to MatMul ?



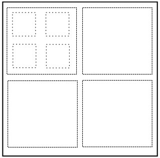
Slab iterative
`LAPACK`



Slab recursive
`FFLAS-FFPACK`



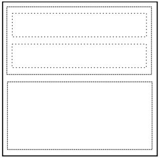
Tile iterative
`PLASMA`



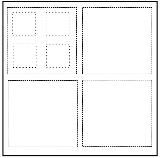
Tile recursive
`FFLAS-FFPACK`

The case of Gaussian elimination

Which reduction to MatMul ?



Slab recursive
`FFLAS-FFPACK`

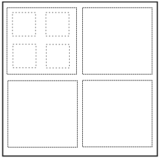


Tile recursive
`FFLAS-FFPACK`

- Sub-cubic complexity: recursive algorithms

The case of Gaussian elimination

Which reduction to MatMul ?



Tile recursive
FFLAS-FFPACK

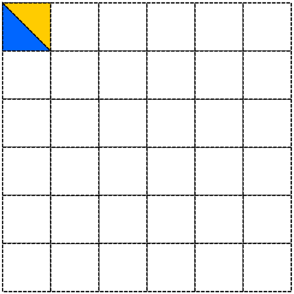
- Sub-cubic complexity: recursive algorithms
- Data locality

Block algorithms

Tile Iterative

Slab Recursive

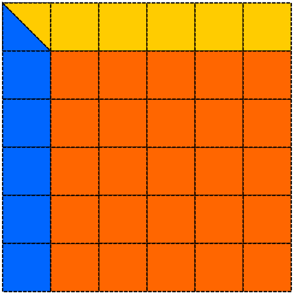
Tile Recursive



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

Block algorithms

Tile Iterative | Slab Recursive | Tile Recursive

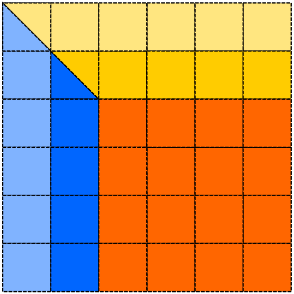


trsm: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

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

Block algorithms

Tile Iterative | Slab Recursive | Tile Recursive



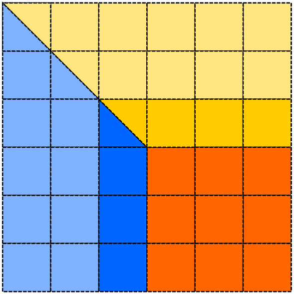
getrf: $A \to L, U$

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

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

Block algorithms

Tile Iterative | Slab Recursive | Tile Recursive



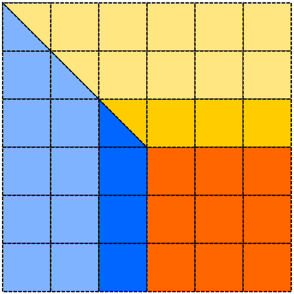
getrf: $A \to L, U$

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

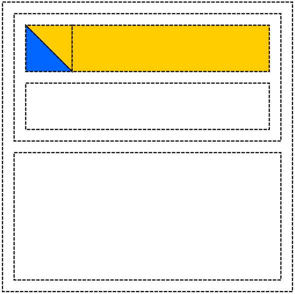
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tile Iterative



Slab Recursive

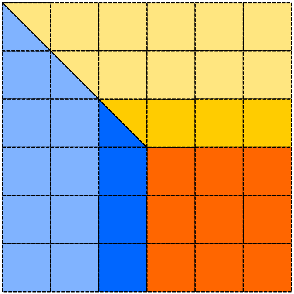


Tile Recursive

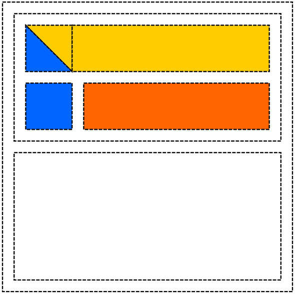
`getrf`: $A \to L, U$

Block algorithms

Tile Iterative



Slab Recursive



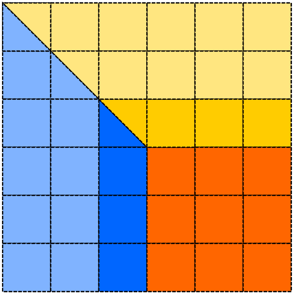
Tile Recursive

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

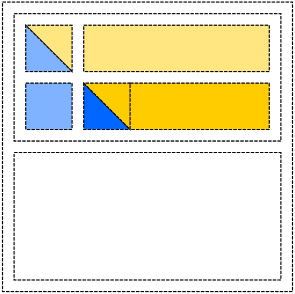
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tile Iterative



Slab Recursive

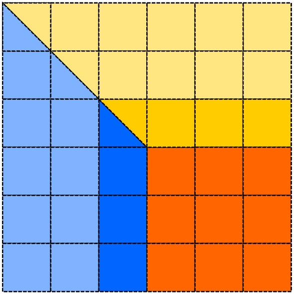


Tile Recursive

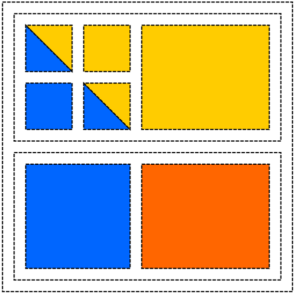
`getrf`: $A \to L, U$

Block algorithms

Tile Iterative



Slab Recursive



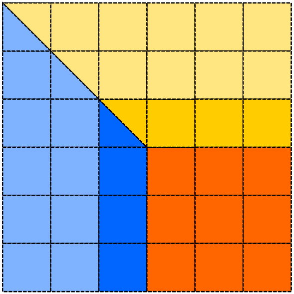
Tile Recursive

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

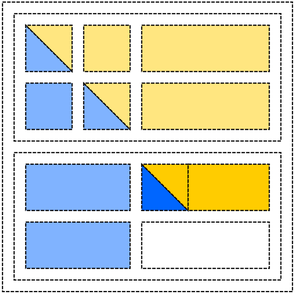
gemm: $C \leftarrow C - A \times B$

Block algorithms

Tile Iterative



Slab Recursive

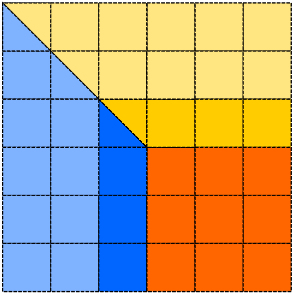


Tile Recursive

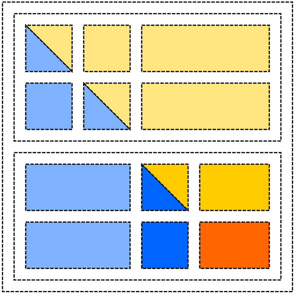
`getrf`: $A \to L, U$

Block algorithms

Tile Iterative



Slab Recursive



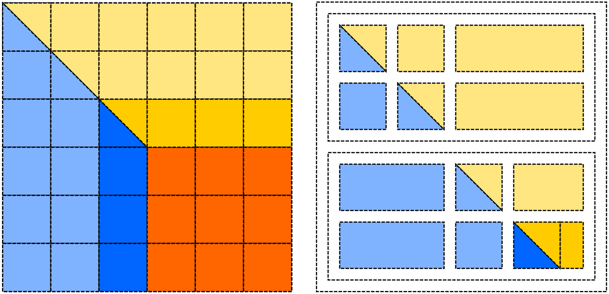
Tile Recursive

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

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

Block algorithms

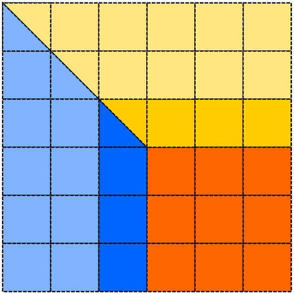
Tile Iterative | Slab Recursive | Tile Recursive



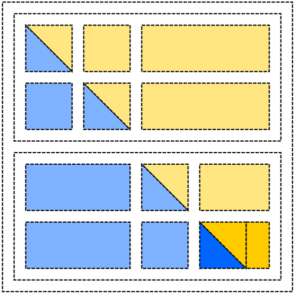
getrf: $A \to L, U$

Block algorithms

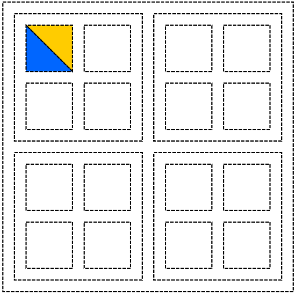
Tile Iterative



Slab Recursive



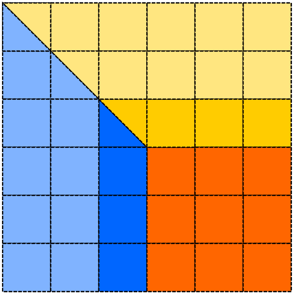
Tile Recursive



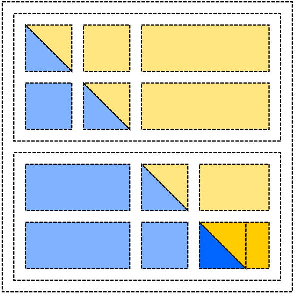
getrf: $A \to L, U$

Block algorithms

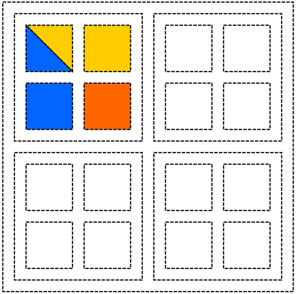
Tile Iterative



Slab Recursive



Tile Recursive

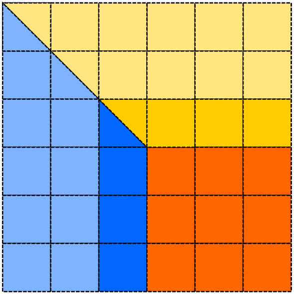


`trsm`: $B \leftarrow BU^{-1}$, $B \leftarrow L^{-1}B$

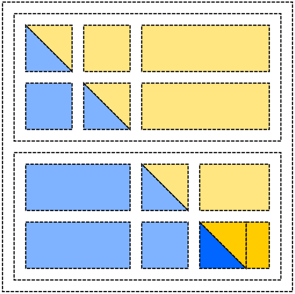
`gemm`: $C \leftarrow C - A \times B$

Block algorithms

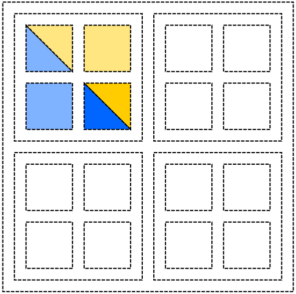
Tile Iterative



Slab Recursive

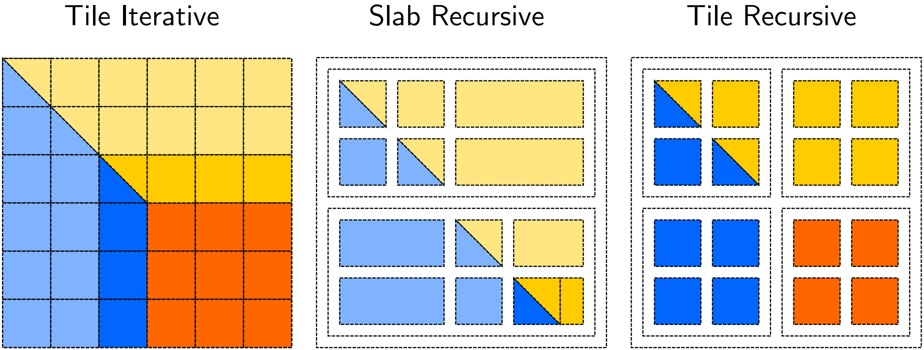


Tile Recursive



getrf: $A \to L, U$

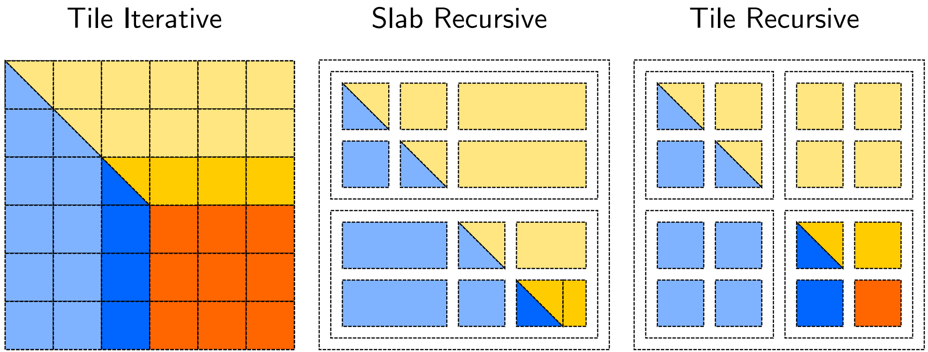
Block algorithms



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

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

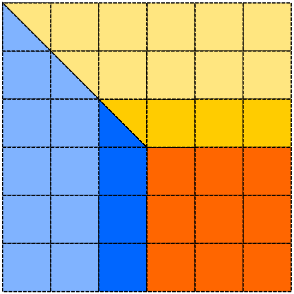
Block algorithms



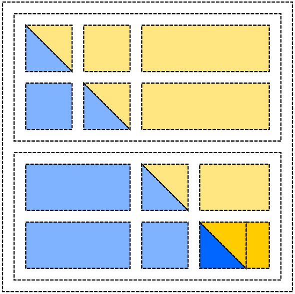
getrf: $A \rightarrow L, U$
trsm: $B \leftarrow BU^{-1}, B \leftarrow L^{-1}B$
gemm: $C \leftarrow C - A \times B$

Block algorithms

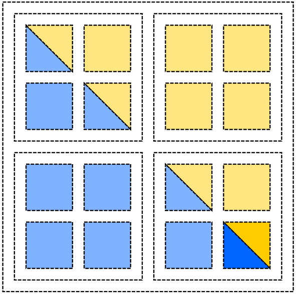
Tile Iterative



Slab Recursive



Tile Recursive



getrf: $A \to L, U$

Counting Modular Reductions

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

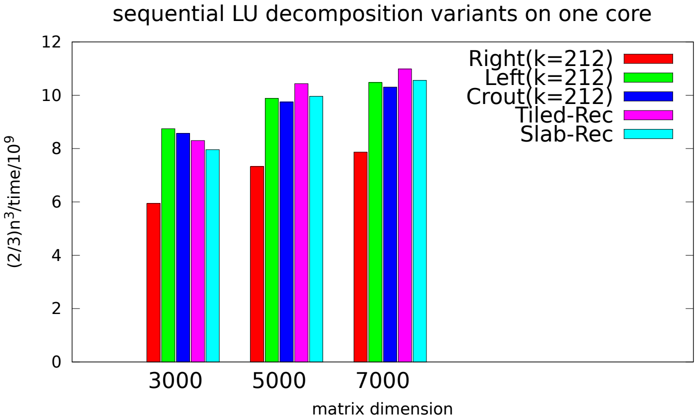
Counting Modular Reductions

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

Counting Modular Reductions

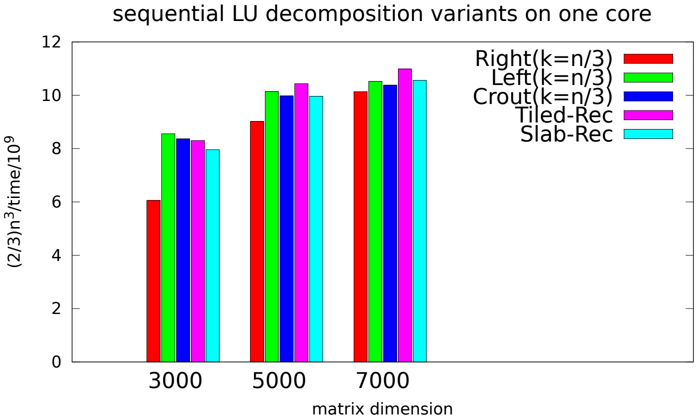
$k \geq 1$	Tile Iter. Right looking	$\frac{\mathbf{1}}{\mathbf{3k}}\mathbf{n^3} + \left(1 - \frac{1}{k}\right) n^2 + \left(\frac{1}{6}k - \frac{5}{2} + \frac{3}{k}\right) n$
	Tile Iter. Left looking	$\left(\mathbf{2} - \frac{\mathbf{1}}{\mathbf{2k}}\right) \mathbf{n^2} + \left(-\frac{5}{2}k - 1 + \frac{2}{k}\right)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

Impact in practice



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

Impact in practice

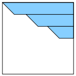


- As anticipated : Right-looking $<$ Crout $<$ Left-looking
- Recursive algorithms stand out with large matrices (Strassen's multiplication) despite their worse mod. reduction complexity.

Dealing with rank deficiencies and computing rank profiles

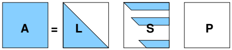
Rank profiles: first linearly independent columns

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

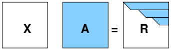


Gaussian elimination revealing echelon forms:

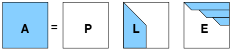
[Ibarra, Moran and Hui 82]



[Keller-Gehrig 85]



[Jeannerod, P. and Storjohann 13]



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

Computing rank profiles

Lessons learned (or what we thought was necessary):

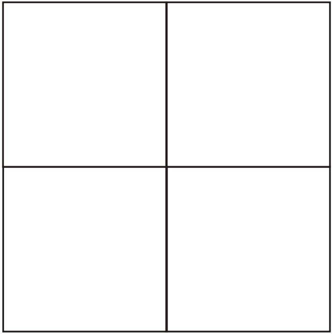
- treat rows in order
- exhaust all columns before considering the next row
- **slab** block splitting required (recursive or iterative)
  $\rightsquigarrow$ similar to partial pivoting

Tile 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 tile recursive algorithm

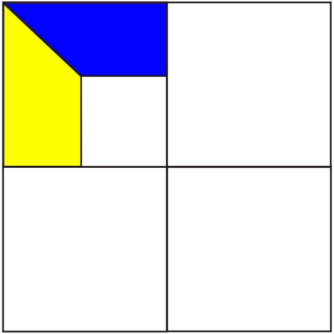
[Dumas, P. and Sultan 13]



2×2 block splitting

A tile recursive algorithm

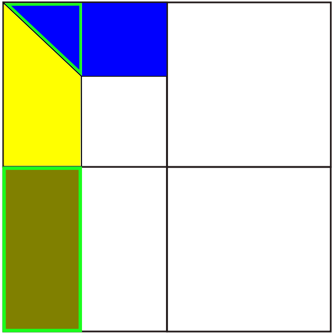
[Dumas, P. and Sultan 13]



Recursive call

A tile recursive algorithm

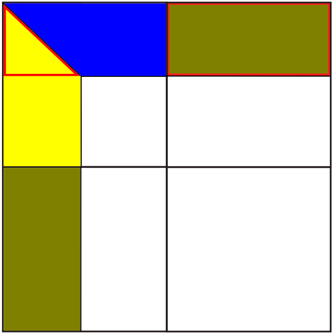
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

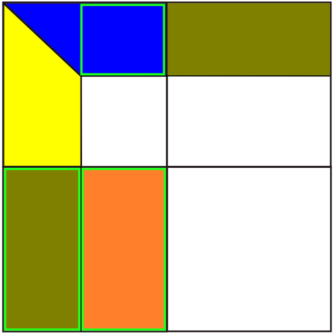
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

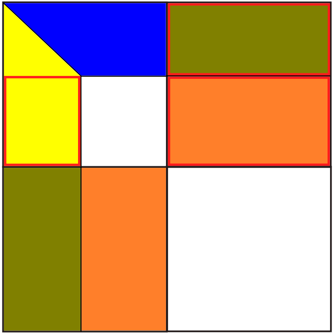
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

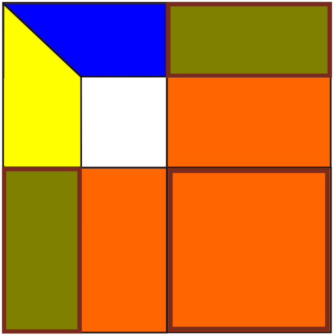
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

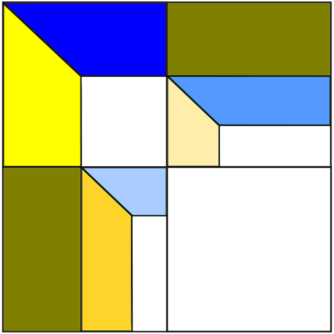
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

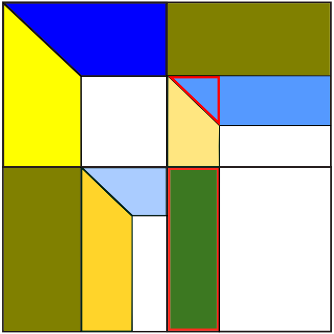
[Dumas, P. and Sultan 13]



2 independent recursive calls

A tile recursive algorithm

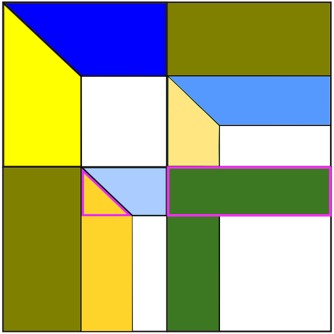
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

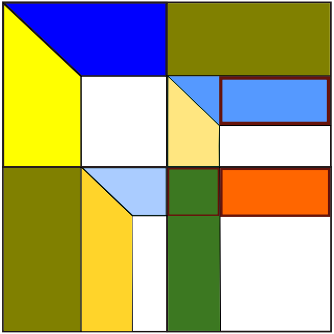
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

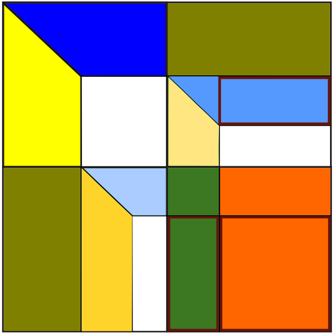
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

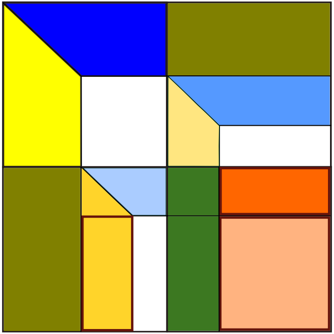
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

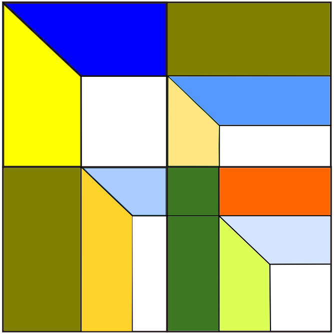
[Dumas, P. and Sultan 13]



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

A tile recursive algorithm

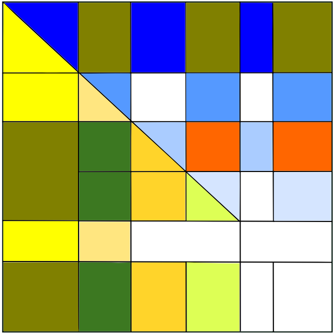
[Dumas, P. and Sultan 13]



Recursive call

A tile recursive algorithm

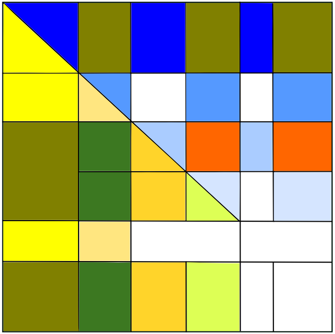
[Dumas, P. and Sultan 13]



Puzzle game (block cyclic rotations)

A tile 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

Outline

1. Choosing the underlying arithmetic
   - Using boolean arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Reductions and building blocks
   - A building block: matrix multiplication
   - Reductions to matrix multiplication
3. Size dimension trade-offs

Size Dimension trade-offs

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

Multimodular methods

over K[X]: evaluation-interpolation

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

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

✓ 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

Cost = Algebraic Cost × Size(Output)

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

Example

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

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

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

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

✓ avoids coefficient blow-up

✗ uniform (worst case) cost for all arithmetic ops

Example

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

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

Size Dimension trade-offs

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

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

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

Size Dimension trade-offs

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

Lifting techniques

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

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

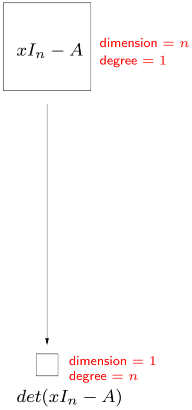
High order lifting : [Storjohann 02,03]

- Fewer iteration steps
- larger dimension in the lifting

Example

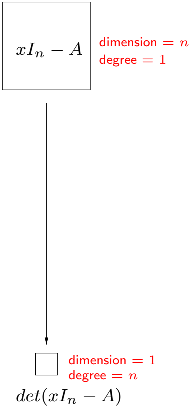
$\texttt{LinSys}_{\mathbb{Z}}(n) = O\tilde{}(n^\omega \log \|A\|_\infty)$

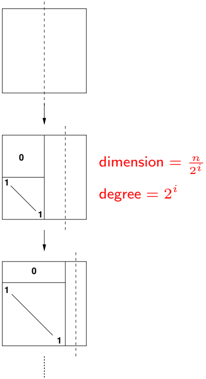
Size dimension trade-offs: the case of the charpoly



Size dimension trade-offs: the case of the charpoly

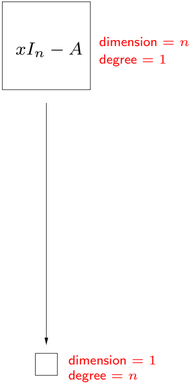
Keller-Gehrig 85





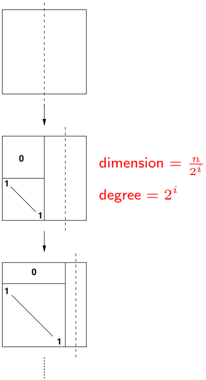
$$\sum_{i=1}^{\log n} n \left(\frac{n}{2^i}\right)^{\omega-1}$$

Size dimension trade-offs: the case of the charpoly



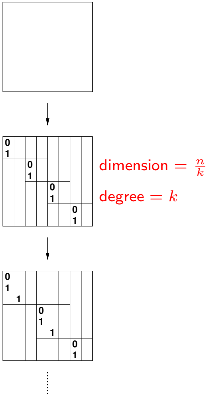
$det(xI_n - A)$

Keller-Gehrig 85



$$\sum_{i=1}^{\log n} n \left(\frac{n}{2^i}\right)^{\omega-1}$$

P. & Storjohann 07



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

Size dimension compromises: the case of charpoly

Recent advances [Neiger, P. 21]

Finally a deterministic $O(n^\omega)$ algorithm

- based on polynomial matrix computations
  - reduced, weak Popov and Popov forms
  - harness recent shifts and partial linearization techniques
- applied to the characteristic matrix (of degree 1)

Size dimension compromises: the case of charpoly

Recent advances [Neiger, P. 21]

Finally a deterministic $O(n^\omega)$ algorithm

- based on polynomial matrix computations
  - reduced, weak Popov and Popov forms
  - harness recent shifts and partial linearization techniques
- applied to the characteristic matrix (of degree 1)

3 types of size dimension compromises for charpoly

KG 85	$C(n,k) = 2C(\frac{n}{2}, 2k) + O(n^\omega k)$	$O(n^\omega \log n)$	determ.
PS 07	$C(n,k) = C(n\frac{k}{k+1}, k+1) + O(n^\omega k)$	$O(n^\omega)$	probab.
NP 21	$C(n,k) = 2C(\frac{n}{2}, k) + C(\frac{n}{2}, 2k) + O(n^\omega M'(k))$	$O(n^\omega)$	determ.

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
- Reductions to MatMul for both

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
- Reductions to MatMul for both
  - asymptotic exponent

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
- Reductions to MatMul for both
  - asymptotic exponent
  - efficiency in practice

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
- Reductions to MatMul for both
  - asymptotic exponent
  - efficiency in practice
- Favor **tile recursive** algorithms

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
- Reductions to MatMul for both
  - asymptotic exponent
  - efficiency in practice
- Favor **tile recursive** algorithms
- **Blocking** made **free** from **pivoting** constraints

Conclusion

Design framework for high performance exact linear algebra

Asymptotic reduction > algorithm tuning > building block implementation

- So far, **floating point** arithmetic delivers best speed
- Reductions to MatMul for both
  - asymptotic exponent
  - efficiency in practice
- Favor **tile recursive** algorithms
- **Blocking** made **free** from **pivoting** constraints
- Seek **size-dimension** trade-offs,

Perspectives

Structured linear algebra

- A lot of action recently [Jeannerod & Al. 08,17], [Villard'18]
  - rank displacement structures
  - quasi-separable structures
- New building blocks identified
- New reduction trees to be built

Perspectives

Structured linear algebra

- A lot of action recently [Jeannerod & Al. 08,17], [Villard'18]
  - rank displacement structures
  - quasi-separable structures
- New building blocks identified
- New reduction trees to be built

Dense linear algebra over $K[X]$

- Tremendous progresses over the last decade: [Neiger, Storjohann, Villard]
- Reduction trees stabilizing
- High performance implementation is still lagging behind

Perspectives

Structured linear algebra

- A lot of action recently [Jeannerod & Al. 08,17], [Villard'18]
  - rank displacement structures
  - quasi-separable structures
- New building blocks identified
- New reduction trees to be built

Dense linear algebra over $K[X]$

- Tremendous progresses over the last decade: [Neiger, Storjohann, Villard]
- Reduction trees stabilizing
- High performance implementation is still lagging behind

Thank you