Computing the Rank profile matrix (and some bonus)

Jounées Nationales du Calcul Formel

Clément Pernet

Laboratoire de l'Informatique du Parallélisme,
Univ. Grenoble Alpes, Univ. de Lyon, CNRS, Inria

Cluny.
2 novembre 2015

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. GMP `mpz_t` (hence `uint64_t`)

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. `GMP mpz_t` (hence `uint64_t`)

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. GMP `mpz_t` (hence `uint64_t`)

Rank profiles: how to select the first 3 linearly indep rows of

$$\begin{array}{|ccccc|}
\hline
0 & 0 & 0 & 0 & 1 \\
0 & 2 & 2 & 0 & 0 \\
0 & 1 & 1 & 1 & 2 \\
1 & 2 & 1 & 2 & 1 \\
0 & 1 & 0 & 0 & 1 \\
\hline
\end{array}$$

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. `GMP mpz_t` (hence `uint64_t`)

Rank profiles: how to select the first 3 linearly indep rows of

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. GMP `mpz_t` (hence `uint64_t`)

Rank profiles: how to select the first 3 linearly indep rows of

$$\begin{array}{|ccccc|}
\hline
0 & 0 & 0 & 0 & 1 \\
0 & 2 & 2 & 0 & 0 \\
0 & 1 & 1 & 1 & 2 \\
1 & 2 & 1 & 2 & 1 \\
\hline
0 & 1 & 0 & 0 & 1 \\
\hline
\end{array}$$

Trailer

Which CPU arithmetic to multiply 2000×2000 matrices over 200bit integers ?

1. boolean
2. `int32_t`
3. `int64_t`
4. `float`
5. `double`
6. GMP `mpz_t` (hence `uint64_t`)

Rank profiles: how to select the first 3 linearly indep rows of

$$\left[\begin{array}{ccccc} 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 2 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \\ \hline 1 & 2 & 1 & 2 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Outline

1. Choosing the underlying arithmetic
   - Using machine word arithmetic
   - Larger field sizes

2. Reductions and building blocks

3. Gaussian elimination
   - Which reduction
   - Computing rank profiles
   - Algorithmic instances
   - Relation to other decompositions
   - The small rank case

Outline

1. Choosing the underlying arithmetic
    - Using machine word arithmetic
    - Larger field sizes

2. Reductions and building blocks

3. Gaussian elimination
    - Which reduction
    - Computing rank profiles
    - Algorithmic instances
    - Relation to other decompositions
    - The small rank case

Most common operation

Most of dense 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, ...

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

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

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 $|ab + y| < 2^{63} \rightsquigarrow |a|, |b| < 2^{31}$

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 $|ab+y| < 2^{63} \rightsquigarrow |a|,|b| < 2^{31}$

```
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 $|ab+y| < 2^{53} \rightsquigarrow |a|,|b| < 2^{26}$

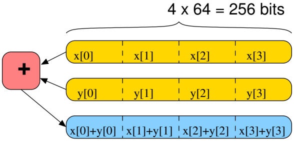
```
movsd    (%rax,%rdx,8), %xmm0      vinsertf128   $0x1, 16(%rcx,%rax), %ymm0,
mulsd    -56(%rbp), %xmm0          vmulpd        %ymm1, %ymm0, %ymm0
addsd    %xmm0, %xmm1              vaddpd        (%rsi,%rax),%ymm0, %ymm0
movq     %xmm1, %rax               vmovapd       %ymm0, (%rsi,%rax)
```

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

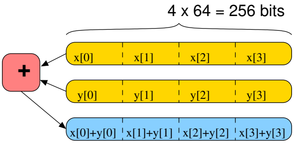


Exploiting *in-core* parallelism

Ingredients

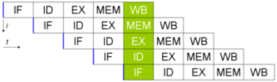
SIMD: Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

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



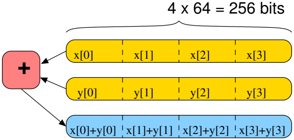
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Exploiting *in-core* parallelism

Ingredients

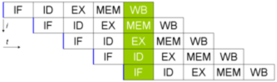
SIMD: Single Instruction Multiple Data:
1 arith. unit acting on a vector of data

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

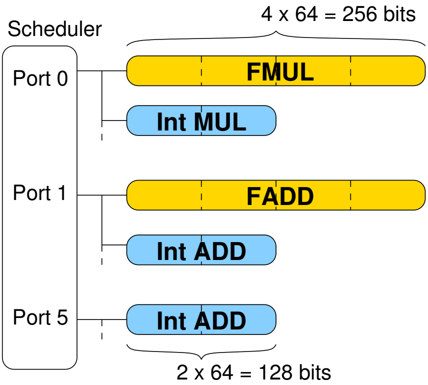


IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

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

SIMD and vectorization

Intel Sandybridge micro-architecture



Performs at every clock cycle:

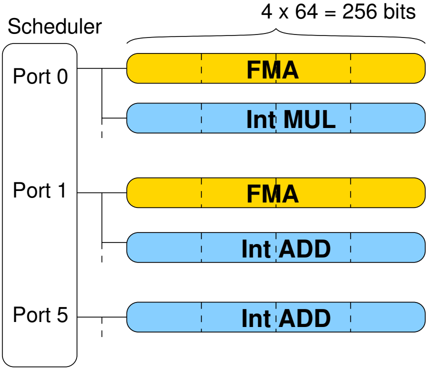
- 1 Floating Pt. Mul × 4
- 1 Floating Pt. Add × 4

Or:

- 1 Integer Mul × 2
- 2 Integer Add × 2

SIMD and vectorization

Intel Haswell micro-architecture



Performs at every clock cycle:

- 2 Floating Pt. Mul & Add × 4

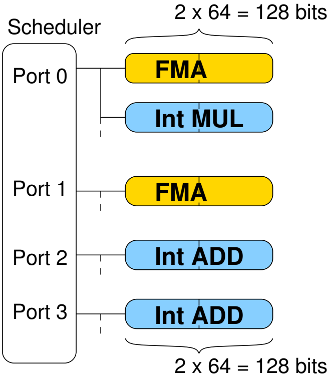
Or:

- 1 Integer Mul × 4
- 2 Integer Add × 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 × 2

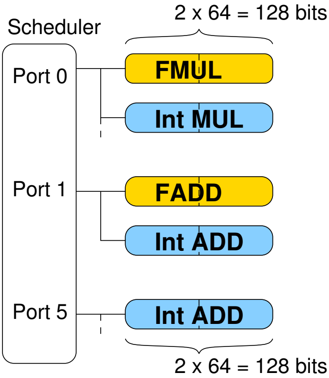
Or:

- 1 Integer Mul × 2
- 2 Integer Add × 2

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

SIMD and vectorization

Intel Nehalem micro-architecture



Performs at every clock cycle:

- 1 Floating Pt. Mul × 2
- 1 Floating Pt. Add × 2

Or:

- 1 Integer Mul × 2
- 2 Integer Add × 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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / 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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / 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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / Cycle) ×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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / Cycle) ×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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / 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	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								
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq $\times$ (# daxpy / 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	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**	
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq × (# daxpy / 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	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**
AMD K10	INT								
SSE4a	FP								

Speed of light: CPU freq $\times$ (# daxpy / 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	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**
AMD K10	INT	64	2	1		1	2.4	**4.8**	
SSE4a	FP	128	1	1		2	2.4	**9.6**	

Speed of light: CPU freq × (# daxpy / Cycle) ×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	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**
AMD K10	INT	64	2	1		1	2.4	**4.8**	
SSE4a	FP	128	1	1		2	2.4	**9.6**	**8.93**

Speed of light: CPU freq × (# daxpy / 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`]

Looking into the near future

Intel Skylake & Knights Landing: AVX512-F

2016 (2017 on Xeons)

- Enlarge SIMD register width to 512 bits (8 `double` or `int64_t`)
- same micro arch : FMA for FP and seprate mul/add for INT.

Looking into the near future

Intel Skylake & Knights Landing: AVX512-F

2016 (2017 on Xeons)

- Enlarge SIMD register width to 512 bits (8 `double` or `int64_t`)
- same micro arch : FMA for FP and seprate mul/add for INT.

Cannonlake: AVX512-IFMA

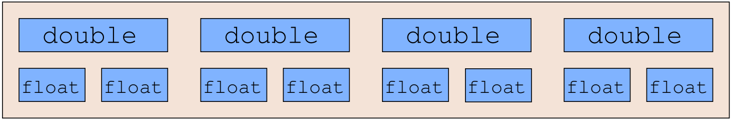
$>$2017

- AVX512 extension: IFMA (Integer FMA): `y += a*b` on `int64_t`
- But limited to the lower 52 bits of the output (uses the FP FMA)

$\rightsquigarrow$ no advantage for `int64_t` over `double`

Integer Packing

32 bits: half the precision twice the speed

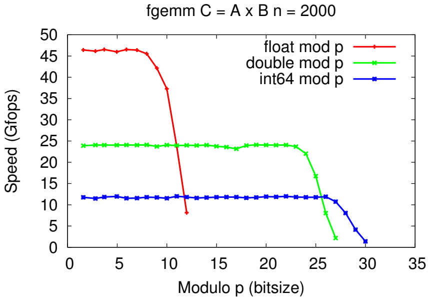


double double double double

float float float float float float float float

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

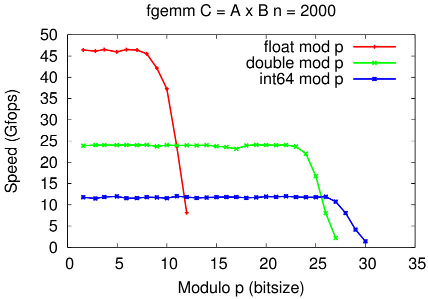


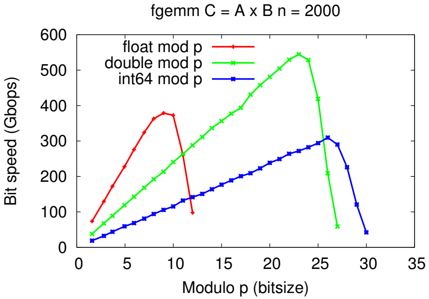
SandyBridge i5-3320M@3.3Ghz. $n = 2000$.

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

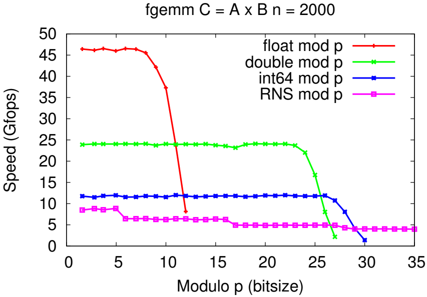
Which integer arithmetic?

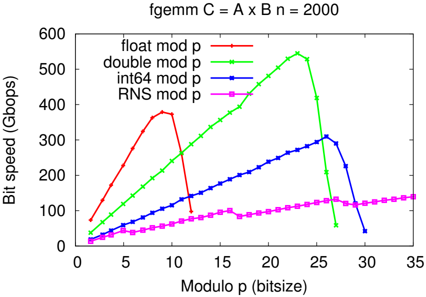
1. big integers (GMP)
2. fixed size multiprecision (Givaro-RecInt)
3. Residue Number Systems (Chinese Remainder theorem)
   $\rightsquigarrow$ using moduli delivering optimum bitspeed

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

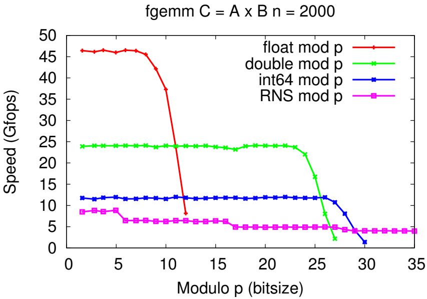
$n = 1000$, on an Intel SandyBridge.

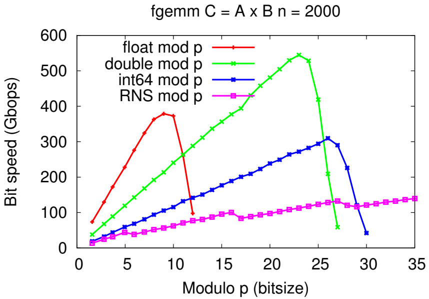
In practice

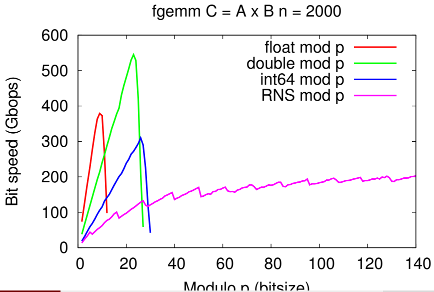

fgemm C = A x B n = 2000
Speed (Gfops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p
0
5
10
15
20
25
30
35
40
45
50
0
5
10
15
20
25
30
35


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
0
100
200
300
400
500
600
0
5
10
15
20
25
30
35

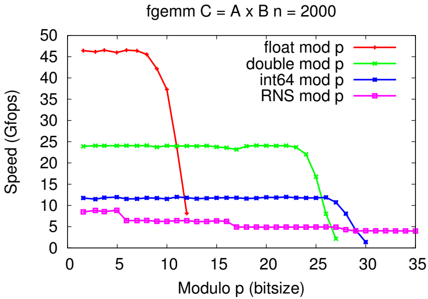
In practice

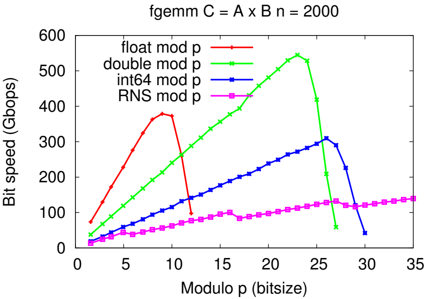

fgemm C = A x B n = 2000
Speed (Gfops)
0
5
10
15
20
25
30
35
40
45
50
Modulo p (bitsize)
0
5
10
15
20
25
30
35
float mod p
double mod p
int64 mod p
RNS mod p

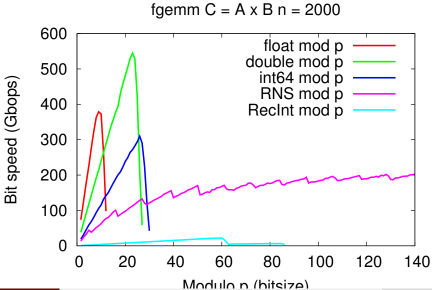

fgemm C = A x B n = 2000
Bit speed (Gbops)
0
100
200
300
400
500
600
Modulo p (bitsize)
0
5
10
15
20
25
30
35
float mod p
double mod p
int64 mod p
RNS mod p


fgemm C = A x B n = 2000
Bit speed (Gbops)
0
100
200
300
400
500
600
Modulo p (bitsize)
0
20
40
60
80
100
120
140
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
0
5
10
15
20
25
30
35
40
45
50
0
5
10
15
20
25
30
35


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
0
100
200
300
400
500
600
0
5
10
15
20
25
30
35


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
0
100
200
300
400
500
600
0
20
40
60
80
100
120
140

Outline

1. Choosing the underlying arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Reductions and building blocks
3. Gaussian elimination
   - Which reduction
   - Computing rank profiles
   - Algorithmic instances
   - Relation to other decompositions
   - The small rank case

Reductions to building blocks

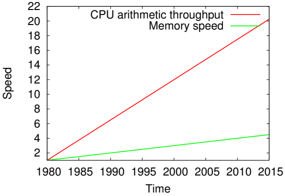
Huge number of algorithmic variants for a given computation.

$\rightsquigarrow$ Need to structure the design for a set of routines :

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

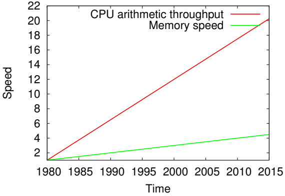
Memory access pattern

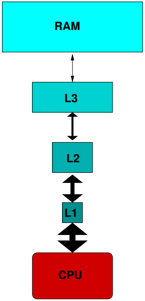
- **The memory wall:** communication speed improves slower than arithmetic



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

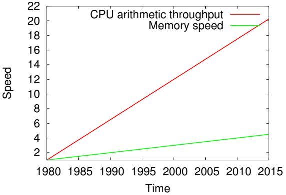
⇝ Need to overlap communications by computation

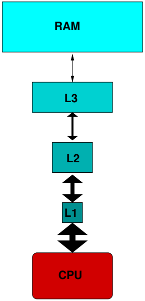
Design of BLAS 3 [Dongarra & Al. 87]

- Group all ops in Matrix products `gemm`:

  Work $O(n^3) >> $ 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]	$O(n^{2.3728639})$

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]	$O(n^{2.3728639})$

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, Danilevskĭi, etc)

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

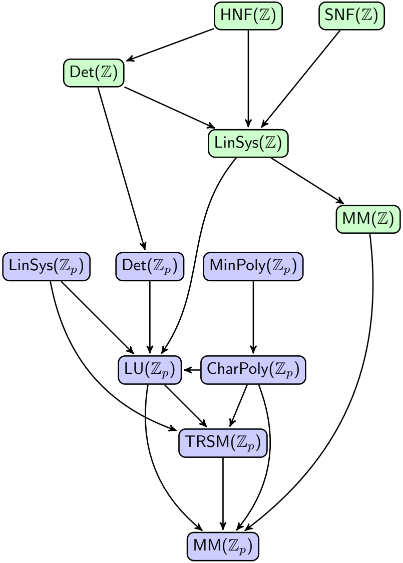
[Strassen 69]:	$O(n^{2.807})$
⋮	
[Schönhage 81]	$O(n^{2.52})$
⋮	
[Coppersmith, Winograd 90]	$O(n^{2.375})$
⋮	
[Le Gall 14]	$O(n^{2.3728639})$

Other operations

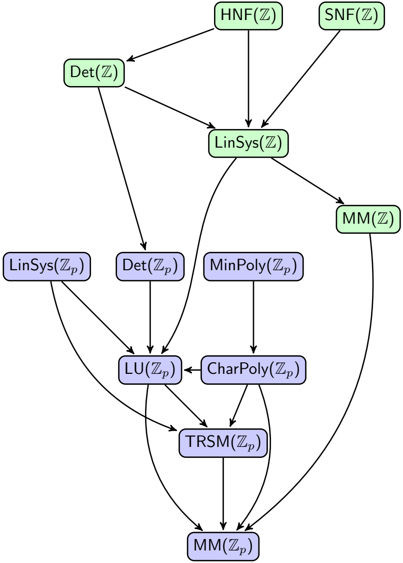
[Strassen 69]:	Inverse in $O(n^\omega)$
[Schönhage 72]:	QR in $O(n^\omega)$
[Bunch, Hopcroft 74]:	LU in $O(n^\omega)$
[Ibarra & *al.* 82]:	Rank in $O(n^\omega)$
[Keller-Gehrig 85]:	CharPoly in $O(n^\omega \log n)$

MatMul has become a building block in theoretical reductions

Reductions: theory



Reductions: theory

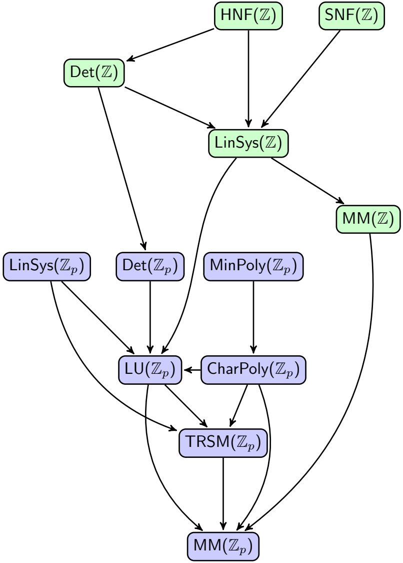


Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Reductions: theory and practice



Common mistrust

Fast linear algebra is

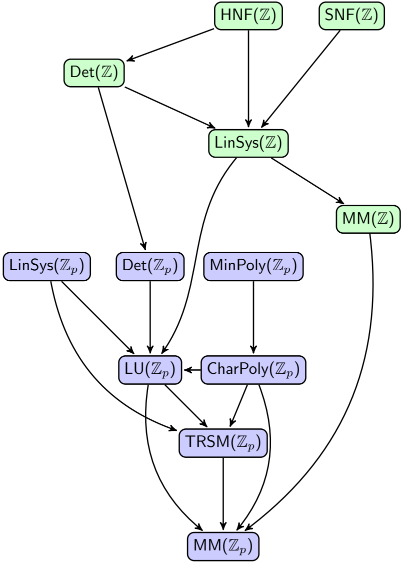
- ✗ never faster
- ✗ numerically unstable

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 $\mathbb{Z}/p\mathbb{Z}$

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

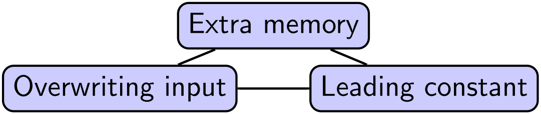
$$\rightsquigarrow \boxed{9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2 < 2^{53}}$$

- Fastest integer arithmetic: `double`
- Cache optimizations

$\rightsquigarrow$ numerical BLAS

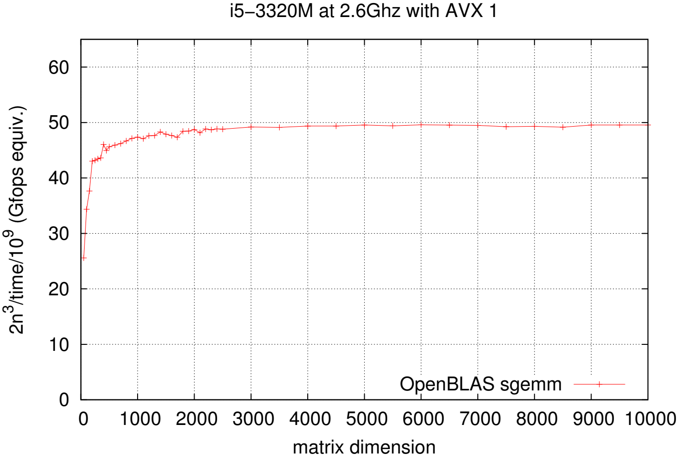
- Strassen-Winograd $6n^{2.807} + \ldots$

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

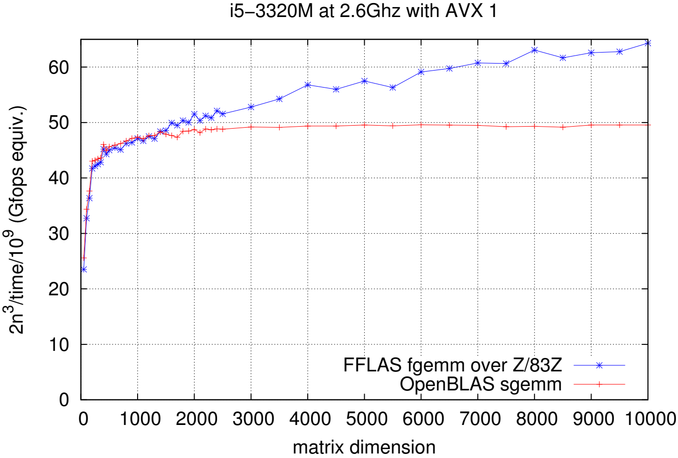


Fully in-place in
$7.2n^{2.807} + \ldots$

Sequential Matrix Multiplication

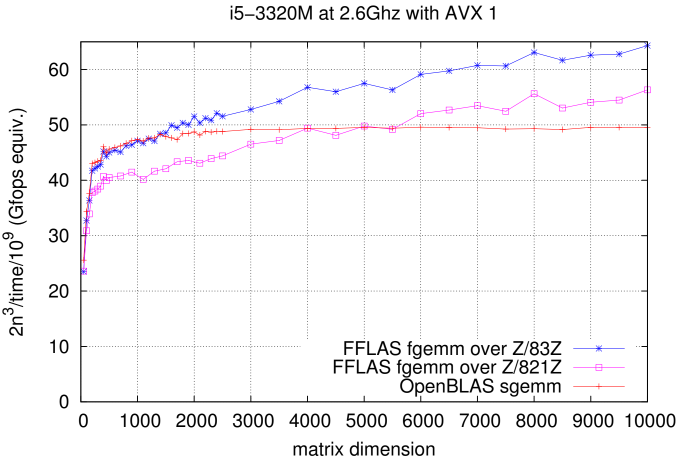


Sequential Matrix Multiplication



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

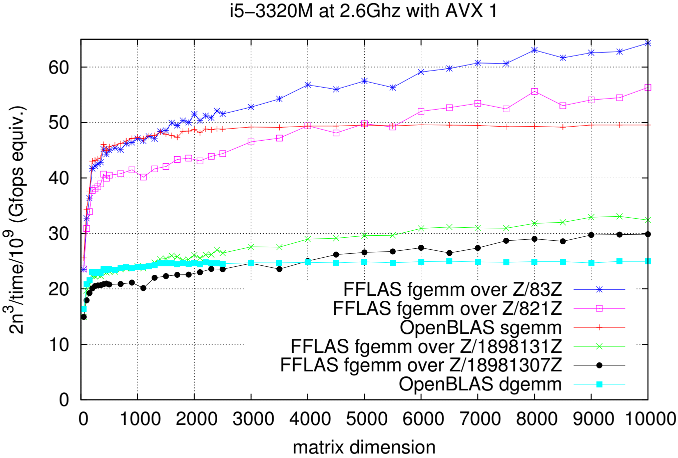
Sequential Matrix Multiplication



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

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

Sequential Matrix Multiplication



$p = 83$, ⇝ 1 mod / 10000 mul.

$p = 821$, ⇝ 1 mod / 100 mul.

$p = 1898131$, ⇝ 1 mod / 10000 mul.

$p = 18981307$, ⇝ 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 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 ⇝ reduces to MatMul ⇝ $O(n^\omega)$

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

×7.63

×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

×7.5

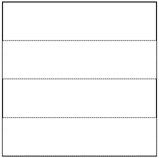
×6.7

Outline

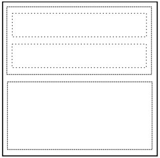
1. Choosing the underlying arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Reductions and building blocks
3. Gaussian elimination
   - Which reduction
   - Computing rank profiles
   - Algorithmic instances
   - Relation to other decompositions
   - The small rank case

The case of Gaussian elimination

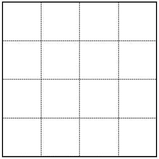
Which reduction to MatMul ?



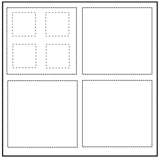
Slab iterative
`LAPACK`



Slab recursive
`FFLAS-FFPACK`



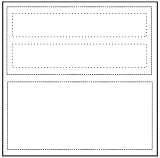
Tile iterative
`PLASMA`



Tile recursive
`FFLAS-FFPACK`

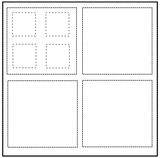
The case of Gaussian elimination

Which reduction to MatMul ?



Slab recursive

FFLAS-FFPACK



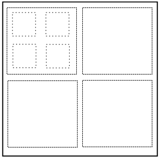
Tile recursive

FFLAS-FFPACK

- Sub-cubic complexity: recursive algorithms

The case of Gaussian elimination

Which reduction to MatMul ?



Tile recursive

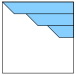
FFLAS-FFPACK

- Sub-cubic complexity: recursive algorithms
- Data locality

Computing rank profiles

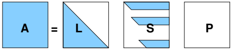
Rank profiles: first linearly independent columns

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

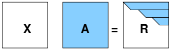


Gaussian elimination revealing echelon forms:

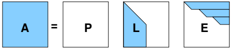
[Ibarra, Moran and Hui 82]



[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

Need for a more flexible 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

Need for a more flexible pivoting

Gathering linear independence invariants

Two ways to look at a matrix: row- or column-wise

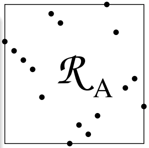
- Row rank profile, column echelon form,
- Column rank profile, row echelon form,

Can't a unique invariant capture all information ?

The rank profile matrix [Dumas, P. and Sultan'15]

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.

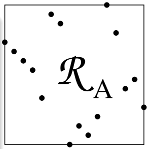


$$\overset{\mathrm{A}}{\boxed{\begin{matrix}1 & 2 & 3 & 4\\ 2 & 4 & 5 & 8\\ 1 & 2 & 3 & 4\\ 3 & 5 & 9 & 12\end{matrix}}} \longrightarrow \overset{\boldsymbol{\mathcal{R}}}{\boxed{\begin{matrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\end{matrix}}}$$

The rank profile matrix [Dumas, P. and Sultan'15]

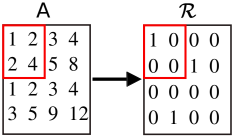
Definition (Rank Profile matrix)

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.*



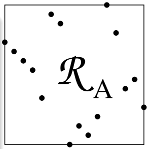
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

RowRP $= \{1\}$
ColRP $= \{1\}$

The rank profile matrix [Dumas, P. and Sultan'15]

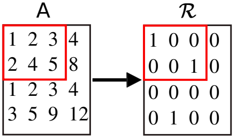
Definition (Rank Profile matrix)

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.*



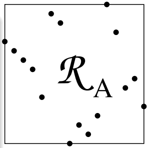
$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{bmatrix} \longrightarrow \mathcal{R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

RowRP = {1,2}
ColRP = {1,3}

The rank profile matrix [Dumas, P. and Sultan'15]

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.*

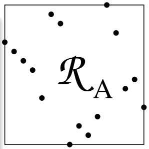
$$\begin{array}{c}\text{A}\\ \left[\begin{array}{cc|cc}1&2&3&4\\2&4&5&8\\1&2&3&4\\3&5&9&12\end{array}\right]\end{array} \longrightarrow \begin{array}{c}\mathcal{R}\\ \left[\begin{array}{cc|cc}1&0&0&0\\0&0&1&0\\0&0&0&0\\0&1&0&0\end{array}\right]\end{array}$$

RowRP $=$ {1,4}
ColRP $=$ {1,2}

The rank profile matrix [Dumas, P. and Sultan'15]

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.*

$$\begin{array}{c} \mathrm{A} \\ \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{array}\right] \end{array} \longrightarrow \begin{array}{c} \mathcal{R} \\ \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

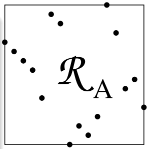
RowRP = {1,4}
ColRP = {1,2}

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

The rank profile matrix [Dumas, P. and Sultan'15]

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.*

$$\begin{array}{ccc} A & & \mathcal{R} \\ \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{array}\right] & \longrightarrow & \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

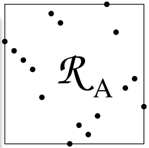
RowRP = {1,4}
ColRP = {1,2}

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

The rank profile matrix [Dumas, P. and Sultan'15]

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.*

$$\overset{A}{\left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{array}\right]} \longrightarrow \overset{\mathcal{R}}{\left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right]}$$

RowRP $=$ {1,4}

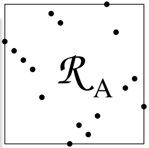
ColRP $=$ {1,2}

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

The rank profile matrix [Dumas, P. and Sultan'15]

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.*

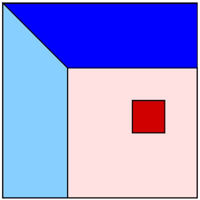
$$\begin{array}{c} A \\ \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 8 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 9 & 12 \end{array}\right] \end{array} \longrightarrow \begin{array}{c} \mathcal{R} \\ \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right] \end{array}$$

RowRP $= \{1,4\}$
ColRP $= \{1,2\}$

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

When does $\boxed{\Pi_{P,Q} = \mathcal{R}(A)}$?

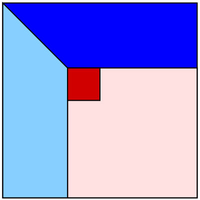
Anatomy of a PLUQ decomposition



Four types of elementary operations:

Search: finding a pivot

Anatomy of a PLUQ decomposition

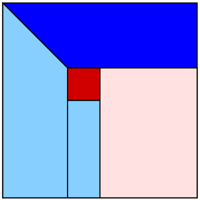


Four types of elementary operations:

Search: finding a pivot

Permutation: moving the pivot to the main diagonal

Anatomy of a PLUQ decomposition



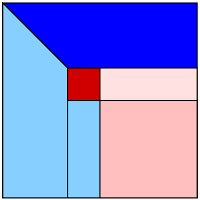
Four types of elementary operations:

Search: finding a pivot

Permutation: moving the pivot to the main diagonal

Normalization: computing L: $l_{i,k} \leftarrow \frac{a_{i,k}}{a_{k,k}}$

Anatomy of a PLUQ decomposition



Four types of elementary operations:

Search: finding a pivot

Permutation: moving the pivot to the main diagonal

Normalization: computing L: $l_{i,k} \leftarrow \frac{a_{i,k}}{a_{k,k}}$

Update: applying the elimination $a_{i,j} \leftarrow a_{i,j} - \frac{a_{i,k}a_{k,j}}{a_{k,k}}$

Impact on the PLUQ decomposition

Normalization: determines whether L or U is unit diagonal

Impact on the PLUQ decomposition

Normalization: determines whether L or U is unit diagonal

Update: no impact on the decomposition, only in the scheduling:

- iterative, tile/slab iterative, recursive,
- left/right looking, Crout

Impact on the PLUQ decomposition

Normalization: determines whether L or U is unit diagonal

Update: no impact on the decomposition, only in the scheduling:

- iterative, tile/slab iterative, recursive,
- left/right looking, Crout

Search: defines the first r values of P and Q

Impact on the PLUQ decomposition

Normalization: determines whether L or U is unit diagonal

Update: no impact on the decomposition, only in the scheduling:

- iterative, tile/slab iterative, recursive,
- left/right looking, Crout

Search: defines the first r values of P and Q

Permutation: impacts all values of P and Q

Impact on the PLUQ decomposition

Normalization: determines whether L or U is unit diagonal

Update: no impact on the decomposition, only in the scheduling:

- iterative, tile/slab iterative, recursive,
- left/right looking, Crout

Search: defines the first r values of P and Q

Permutation: impacts all values of P and Q

Problem (Reformulation)

Under what conditions on the **Search** *and* **Permutation** *operations does a PLUQ decomposition algorithm reveals RowRP, ColRP or $\mathcal{R}_A$?*

The Pivoting matrix

Definition (The pivoting matrix)

Given a PLUQ decomposition $A = PLUQ$ with rank r, define

$$\Pi_{P,Q} = P \begin{bmatrix} I_r & \\ & \end{bmatrix} Q.$$

Locates the position of the pivots in the matrix A.

The Pivoting matrix

Definition (The pivoting matrix)

Given a PLUQ decomposition $A = PLUQ$ with rank r, define

$$\Pi_{P,Q} = P \begin{bmatrix} I_r & \\ & \end{bmatrix} Q.$$

Locates the position of the pivots in the matrix A.

Problem (Rank profile revealing PLUQ decompositions)

Under which conditions

- $\Pi_{P,Q} = \mathcal{R}_A$

The Pivoting matrix

Definition (The pivoting matrix)

Given a PLUQ decomposition $A = PLUQ$ with rank r, define

$$\Pi_{P,Q} = P \begin{bmatrix} I_r & \\ & \end{bmatrix} Q.$$

Locates the position of the pivots in the matrix A.

Problem (Rank profile revealing PLUQ decompositions)

Under which conditions

- $\Pi_{P,Q} = \mathcal{R}_A$
- $RowSupp(\Pi_{P,Q}) = RowSupp(\mathcal{R}_A) = RowRP(A)$ *(Weaker)*
- $ColSupp(\Pi_{P,Q}) = ColSupp(\mathcal{R}_A) = ColRP(A)$ *(Weaker)*

The Search operation

Various strategies depending on the context

Numerical stability: find the absolute largest pivot

Data locality: find pivot not too far from the main diagonal

Sparsity: find pivot that minimizes/reduce fill-in

The Search operation

Various strategies depending on the context

Numerical stability: find the absolute largest pivot

Data locality: find pivot not too far from the main diagonal

Sparsity: find pivot that minimizes/reduce fill-in

Search revealing rank profiles

- No stability issue over exact domains
- Intuition: must **minimize** some **ordering of the row/col indices** (notion of rank profile)

The Search operation

Various strategies depending on the context

Numerical stability: find the absolute largest pivot

Data locality: find pivot not too far from the main diagonal

Sparsity: find pivot that minimizes/reduce fill-in

Search revealing rank profiles

- No stability issue over exact domains
- Intuition: must **minimize** some **ordering of the row/col indices** (notion of rank profile)

Example

Search: "Any non zero element on the topmost row":

$$A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

The Search operation

Various strategies depending on the context

Numerical stability: find the absolute largest pivot

Data locality: find pivot not too far from the main diagonal

Sparsity: find pivot that minimizes/reduce fill-in

Search revealing rank profiles

- No stability issue over exact domains
- Intuition: must **minimize** some **ordering of the row/col indices** (notion of rank profile)

Example

Search: "Any non zero element on the topmost row":

$$A = \begin{bmatrix} 2&0&3&0\\ 1&0&0&0\\ 0&0&4&0\\ 0&2&0&1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1&0&0&0\\ 0&0&1&0\\ 0&0&0&0\\ 0&1&0&0 \end{bmatrix}, \quad \Pi_{P,Q} = \begin{bmatrix} 0&0&1&0\\ 1&0&0&0\\ 0&0&0&0\\ 0&1&0&0 \end{bmatrix}.$$

The Search operation

Various strategies depending on the context

Numerical stability: find the absolute largest pivot

Data locality: find pivot not too far from the main diagonal

Sparsity: find pivot that minimizes/reduce fill-in

Search revealing rank profiles

- No stability issue over exact domains
- Intuition: must **minimize** some **ordering of the row/col indices** (notion of rank profile)

Example

Search: "Any non zero element on the topmost row":

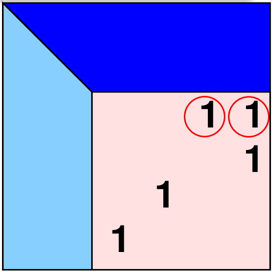
$$A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad \Pi_{P,Q} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \qquad \rightsquigarrow \text{RowRP}=\{1,2,4\}$$

Pivoting and permutation strategies

Pivot Search

Pivot's (i,j) position minimizes some pre-order:

Row order: any non-zero on the first non-zero row

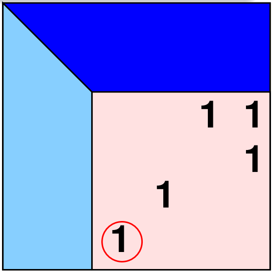


Pivoting and permutation strategies

Pivot Search

Pivot's (i, j) position minimizes some pre-order:

Row/Col order: any non-zero on the first non-zero row/col



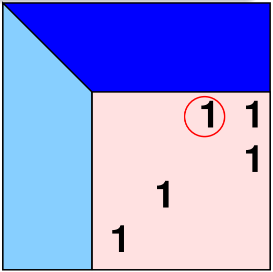
Pivoting and permutation strategies

Pivot Search

Pivot's (i, j) position minimizes some pre-order:

Row/Col order: any non-zero on the first non-zero row/col

Lex order: first non-zero on the first non-zero row



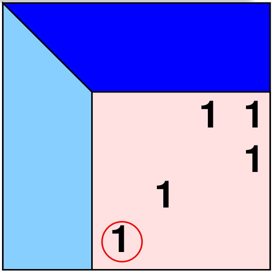
Pivoting and permutation strategies

Pivot Search

Pivot's (i, j) position minimizes some pre-order:

Row/Col order: any non-zero on the first non-zero row/col

Lex/RevLex order: first non-zero on the first non-zero row/col



Pivoting and permutation strategies

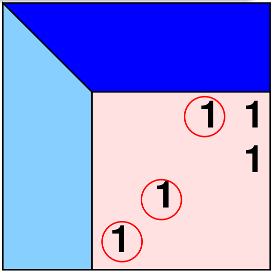
Pivot Search

Pivot's (i, j) position minimizes some pre-order:

Row/Col order: any non-zero on the first non-zero row/col

Lex/RevLex order: first non-zero on the first non-zero row/col

Product order: first non-zero in the (i, j) leading sub-matrix



Sufficient ?

Is lexicographic ordering sufficient to reveal both rank profiles?

Example

With a lexicographic ordering

1. $A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \Pi_{P,Q}$

Sufficient ?

Is lexicographic ordering sufficient to reveal both rank profiles?

Example

With a lexicographic ordering

1. $A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \Pi_{P,Q}$
2. But $A = \begin{bmatrix} 0 & 0 & 1 \\ 2 & 3 & 0 \end{bmatrix} \rightsquigarrow \mathcal{R}_A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ and $\Pi_{P,Q} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

Sufficient ?

Is lexicographic ordering sufficient to reveal both rank profiles?

Example

With a lexicographic ordering

1. $A = \begin{bmatrix} 2 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{bmatrix} \Rightarrow \mathcal{R}_A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \Pi_{P,Q}$
2. But $A = \begin{bmatrix} 0 & 0 & 1 \\ 2 & 3 & 0 \end{bmatrix} \rightsquigarrow \mathcal{R}_A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$ and $\Pi_{P,Q} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$.

$\rightsquigarrow$ Pivot Swaps mix-up precedence between rows/cols.

$\rightsquigarrow$ **Permutations** also have to be considered

Pivoting and permutation strategies

Pivot Search

Pivot's (i,j) position minimizes some pre-order:

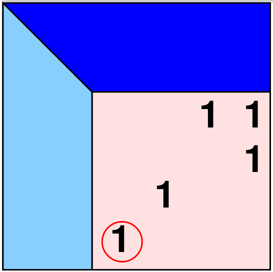
Row/Col order: any non-zero on the first non-zero row/col

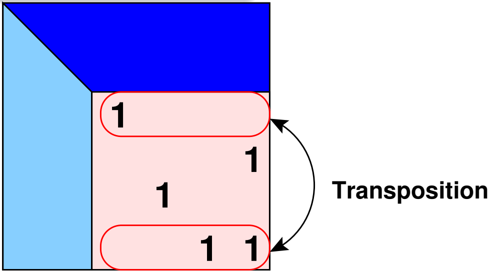
Lex/RevLex order: first non-zero on the first non-zero row/col

Product order: first non-zero in the (i,j) leading sub-matrix

Permutation

- Transpositions





Pivoting and permutation strategies

Pivot Search

Pivot's (i, j) position minimizes some pre-order:

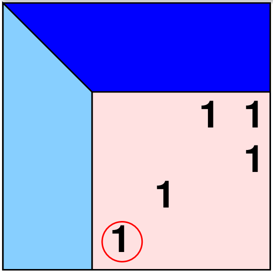
Row/Col order: any non-zero on the first non-zero row/col

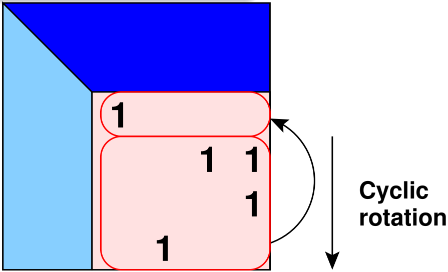
Lex/RevLex order: first non-zero on the first non-zero row/col

Product order: first non-zero in the (i, j) leading sub-matrix

Permutation

- Transpositions
- Cyclic Rotations





Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order Col. order						
Lexico.						
Rev. lex.						
Product						

Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order Col. order	Transposition	Transposition	✓			[IMH82] [JPS13]
Lexico.						
Rev. lex.						
Product						

- RowRP $= \begin{bmatrix}1 & 2 & \dots & m\end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$

Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order	Transposition	Transposition	✓			[IMH82] [JPS13]
Col. order	Transposition	Transposition		✓		[KG85] [JPS13]
Lexico.						
Rev. lex.						
Product						

- RowRP $= \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$
- ColRP $= \begin{bmatrix} I_r & 0 \end{bmatrix} Q \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix}^T$

Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order	Transposition	Transposition	✓			[IMH82] [JPS13]
Col. order	Transposition	Transposition		✓		[KG85] [JPS13]
Lexico.	Transposition	Transposition	✓			[Sto00]
Rev. lex.	Transposition	Transposition		✓		[Sto00]
Product						

- RowRP $= \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$
- ColRP $= \begin{bmatrix} I_r & 0 \end{bmatrix} Q \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix}^T$

Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order	Transposition	Transposition	✓			[IMH82] [JPS13]
Col. order	Transposition	Transposition		✓		[KG85] [JPS13]
Lexico.	Transposition	Transposition	✓			[Sto00]
Rev. lex.	Transposition	Transposition		✓		[Sto00]
Product	Rotation	Rotation	✓	✓	✓	[DPS13]

- RowRP $= \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$
- ColRP $= \begin{bmatrix} I_r & 0 \end{bmatrix} Q \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix}^T$
- $\mathcal{R}_A = P \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} Q$

Pivoting strategies revealing rank profiles

For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order	Transposition	Transposition	✓			[IMH82] [JPS13]
Col. order	Transposition	Transposition		✓		[KG85] [JPS13]
Lexico.	Transposition	Transposition	✓			[Sto00]
Rev. lex.	Transposition	Transposition		✓		[Sto00]
Product	Rotation	Transposition	✓			[DPS15]
Product	Transposition	Rotation		✓		[DPS15]
Product	Rotation	Rotation	✓	✓	✓	[DPS13]

- RowRP $= \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$
- ColRP $= \begin{bmatrix} I_r & 0 \end{bmatrix} Q \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix}^T$
- $\mathcal{R}_A = P \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} Q$

Pivoting strategies revealing rank profiles

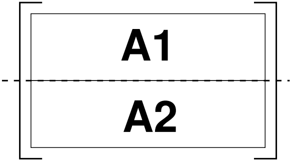
For any type of PLUQ algorithm: iterative / block iterative / recursive

Search	**Row perm.**	**Col. perm.**	RowRP	ColRP	$\mathcal{R}_A$	**Instance**
Row order	Transposition	Transposition	✓			[IMH82] [JPS13]
Col. order	Transposition	Transposition		✓		[KG85] [JPS13]
Lexico.	Transposition	Transposition	✓			[Sto00]
Lexico.	Transposition	Rotation	✓	✓	✓	[DPS15]
Lexico.	Rotation	Rotation	✓	✓	✓	[DPS15]
Rev. lex.	Transposition	Transposition		✓		[Sto00]
Rev. lex.	Rotation	Transposition	✓	✓	✓	[DPS15]
Rev. lex.	Rotation	Rotation	✓	✓	✓	[DPS15]
Product	Rotation	Transposition	✓			[DPS15]
Product	Transposition	Rotation		✓		[DPS15]
Product	Rotation	Rotation	✓	✓	✓	[DPS13]

- RowRP $= \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix} P \begin{bmatrix} I_r \\ 0 \end{bmatrix}$
- ColRP $= \begin{bmatrix} I_r & 0 \end{bmatrix} Q \begin{bmatrix} 1 & 2 & \dots & m \end{bmatrix}^T$
- $\mathcal{R}_A = P \begin{bmatrix} I_r & \\ & 0 \end{bmatrix} Q$

The slab recursive algorithm

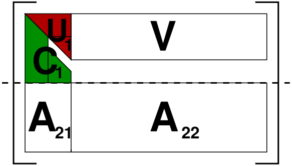
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise

The slab recursive algorithm

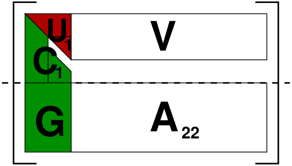
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise
2. Recursive call on A_1

The slab recursive algorithm

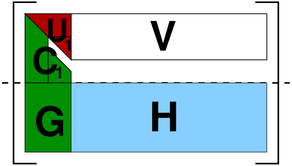
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21}U_1^{-1}$ (`trsm`)

The slab recursive algorithm

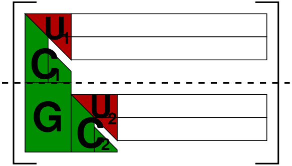
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21}U_1^{-1}$ (`trsm`)
4. $H \leftarrow A_{22} - G \times V$ (`MM`)

The slab recursive algorithm

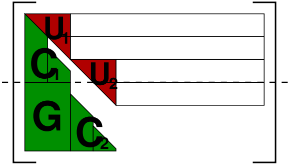
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21}U_1^{-1}$ (`trsm`)
4. $H \leftarrow A_{22} - G \times V$ (`MM`)
5. Recursive call on H

The slab recursive algorithm

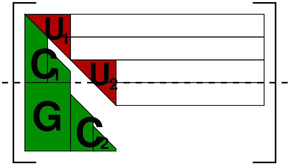
Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21} U_1^{-1}$ (`trsm`)
4. $H \leftarrow A_{22} - G \times V$ (`MM`)
5. Recursive call on H
6. Row permutations

The slab recursive algorithm

Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



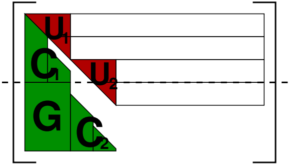
1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21}U_1^{-1}$ (`trsm`)
4. $H \leftarrow A_{22} - G \times V$ (`MM`)
5. Recursive call on H
6. Row permutations

Implements the lexicographic order search.

- Col/Row Transpositions : Computes the ColRP

The slab recursive algorithm

Slab Recursive LU [IMH82, KG85, Sto00, JPS13]



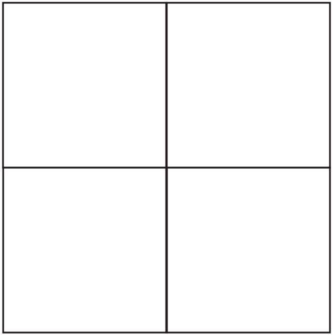
1. Split A Row-wise
2. Recursive call on A_1
3. $G \leftarrow A_{21}U_1^{-1}$ (trsm)
4. $H \leftarrow A_{22} - G \times V$ (MM)
5. Recursive call on H
6. Row permutations

Implements the lexicographic order search.

- Col/Row Transpositions : Computes the ColRP
- Row Rotations : Computes $\mathcal{R}_A$ [DPS15]

The tiled recursive algorithm

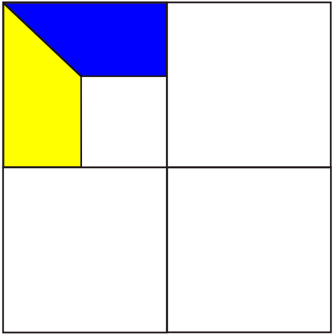
Dumas, P. and Sultan 13



2×2 block splitting

The tiled recursive algorithm

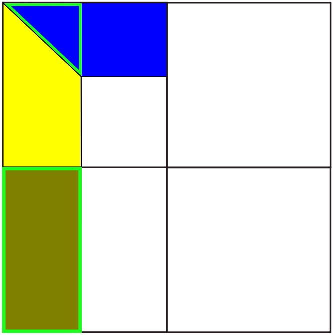
Dumas, P. and Sultan 13



Recursive call

The tiled recursive algorithm

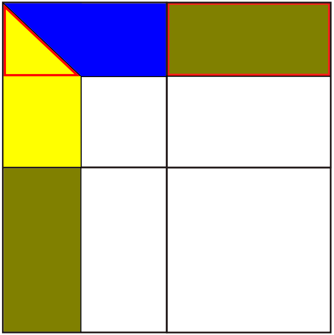
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

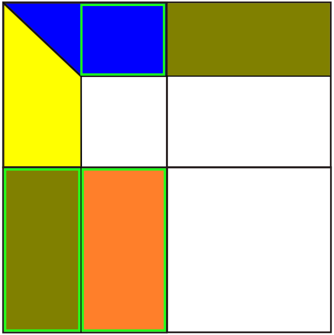
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

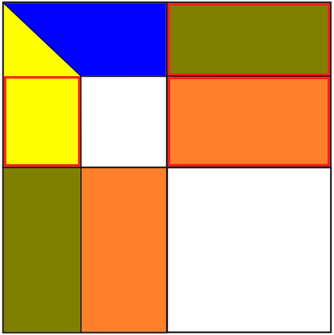
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

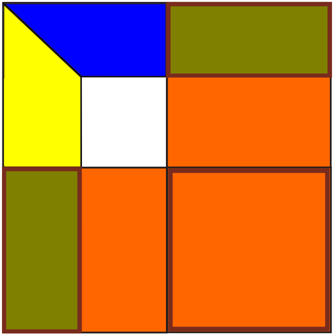
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

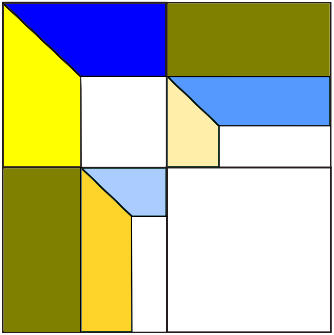
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

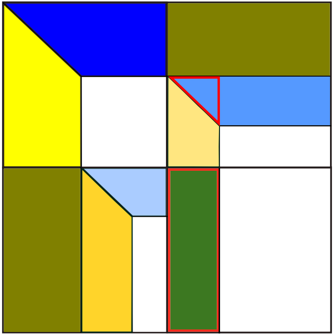
Dumas, P. and Sultan 13



2 independent recursive calls (compatible with the **product order**)

The tiled recursive algorithm

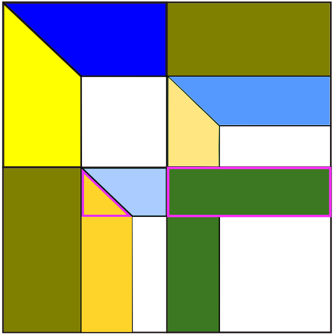
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

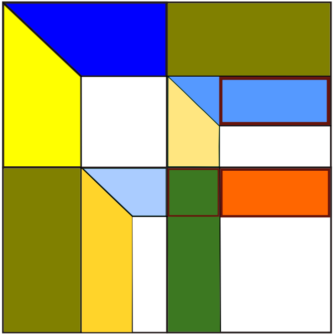
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

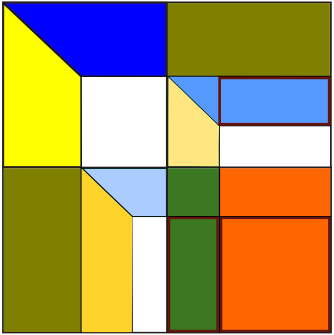
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

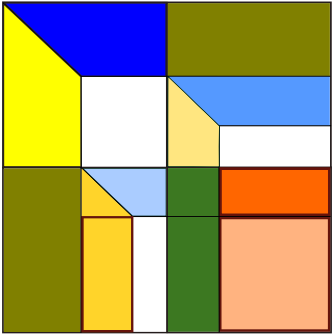
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

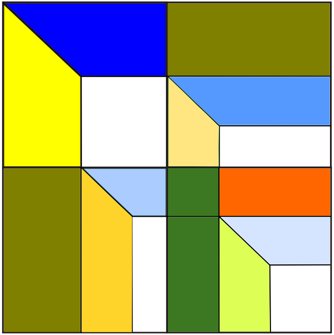
Dumas, P. and Sultan 13



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

The tiled recursive algorithm

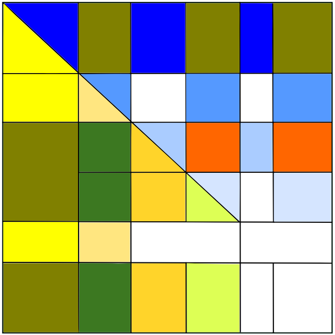
📄 Dumas, P. and Sultan 13



Recursive call

The tiled recursive algorithm

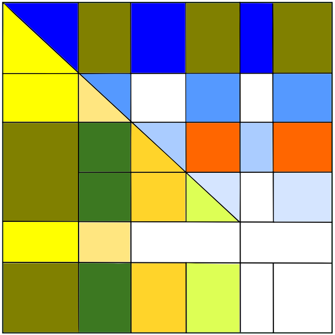
Dumas, P. and Sultan 13



Puzzle game (block **rotations**)

The tiled recursive algorithm

Dumas, P. and Sultan 13



- $O(mnr^{\omega-2})$ ($2/3n^3$ for $\omega = 3$)
- fewer modular reductions than slab algorithms
- rank deficiency introduces parallelism

Iterative algorithms

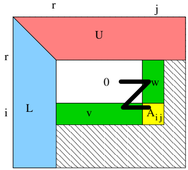
- Unefficient with large problems
- Good for base case implementations (faster in-cache computation)

Iterative algorithms

- Unefficient with large problems
- Good for base case implementations (faster in-cache computation)

Which base case algorithm?

- Formerly [DPS13]: **product order** iterative algorithm
  - ✗ many permutations
  - ✗ many modular reductions

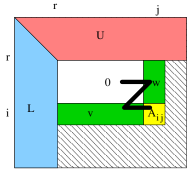


Iterative algorithms

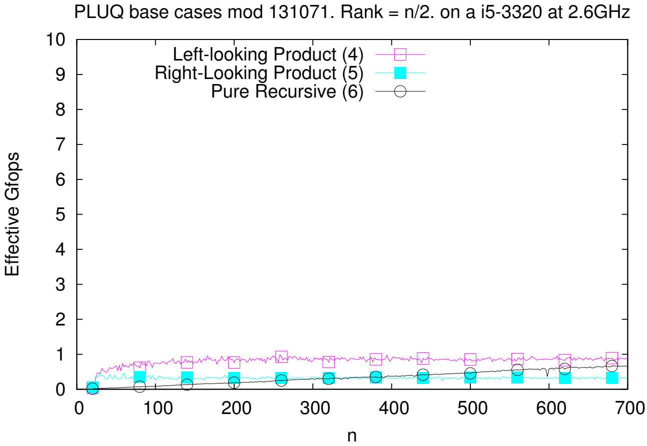
- Unefficient with large problems
- Good for base case implementations (faster in-cache computation)

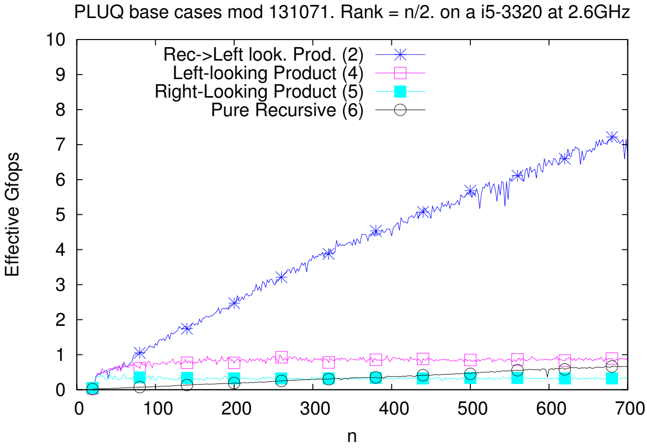
Which base case algorithm?

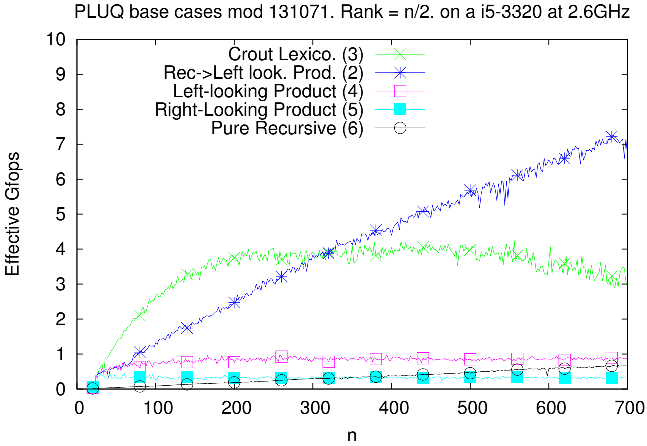
- Formerly [DPS13]: **product order** iterative algorithm
  - ✗ many permutations
  - ✗ many modular reductions

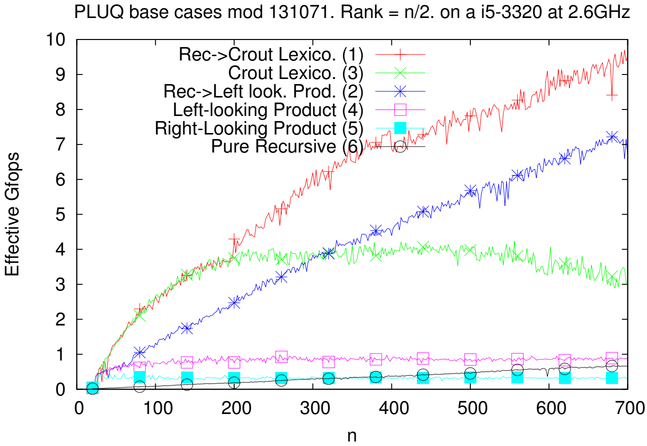


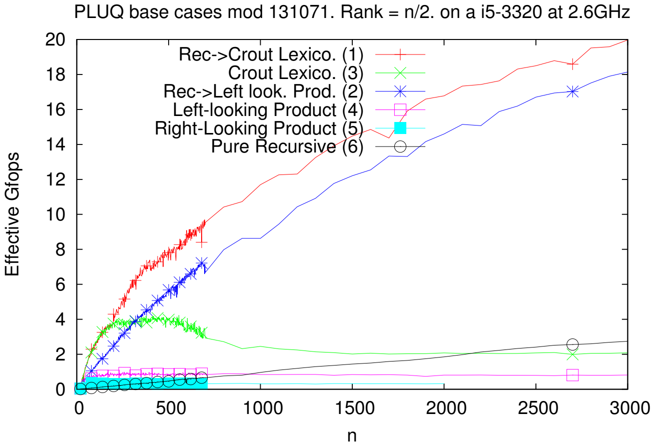
- [DPS15]: Simply use the schoolbook algorithm (Lexico+Rotations)
  - ✓ fewer permutations
  - ✓ modular reductions delayed more easily
  - ✓ Crout variant: better data access pattern

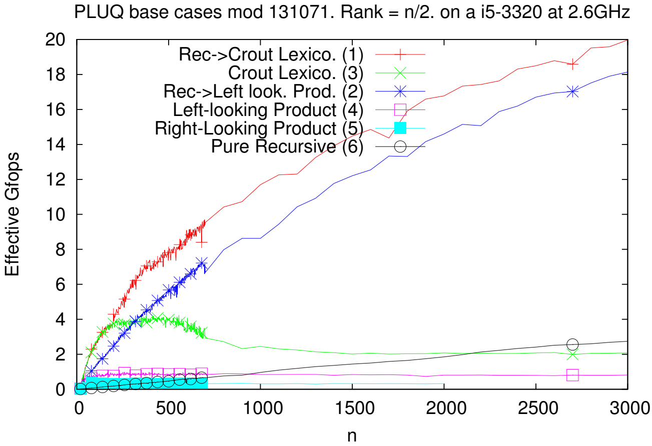

PLUQ base cases mod 131071. Rank = n/2. on a i5-3320 at 2.6GHz
Effective Gfops
0
1
2
3
4
5
6
7
8
9
10
n
0
100
200
300
400
500
600
700
Left-looking Product (4)
Right-Looking Product (5)
Pure Recursive (6)


PLUQ base cases mod 131071. Rank = n/2. on a i5-3320 at 2.6GHz
Effective Gfops
n
Rec->Left look. Prod. (2)
Left-looking Product (4)
Right-Looking Product (5)
Pure Recursive (6)
0
1
2
3
4
5
6
7
8
9
10
0
100
200
300
400
500
600
700


PLUQ base cases mod 131071. Rank = n/2. on a i5-3320 at 2.6GHz
Effective Gfops
n
0
1
2
3
4
5
6
7
8
9
10
0
100
200
300
400
500
600
700
Crout Lexico. (3)
Rec->Left look. Prod. (2)
Left-looking Product (4)
Right-Looking Product (5)
Pure Recursive (6)


PLUQ base cases mod 131071. Rank = n/2. on a i5-3320 at 2.6GHz
Rec->Crout Lexico. (1)
Crout Lexico. (3)
Rec->Left look. Prod. (2)
Left-looking Product (4)
Right-Looking Product (5)
Pure Recursive (6)
Effective Gfops
n
0
1
2
3
4
5
6
7
8
9
10
0
100
200
300
400
500
600
700


PLUQ base cases mod 131071. Rank = n/2. on a i5-3320 at 2.6GHz
Effective Gfops
n
Rec->Crout Lexico. (1)
Crout Lexico. (3)
Rec->Left look. Prod. (2)
Left-looking Product (4)
Right-Looking Product (5)
Pure Recursive (6)
0
2
4
6
8
10
12
14
16
18
20
0
500
1000
1500
2000
2500
3000



- > 2 Gfops improvement
- Implemented in `FFLAS-FFPACK` (kernel of `LinBox`).

LUP and PLU decompositions

LUP

If A has generic RowRP

- $LUP(A)$ with Lex order and col. rot.: $\rightsquigarrow \begin{matrix} I_r & \\ & 0 \end{matrix} P = \mathcal{R}_A$

In particular, if A has full row rank and $m = n$: $\rightsquigarrow P = \mathcal{R}_A$

LUP and PLU decompositions

LUP

If A has generic RowRP

- $LUP(A)$ with Lex order and col. rot.: $\rightsquigarrow \begin{matrix} I_r & \\ & 0 \end{matrix} P = \mathcal{R}_A$

In particular, if A has full row rank and $m = n$: $\rightsquigarrow P = \mathcal{R}_A$

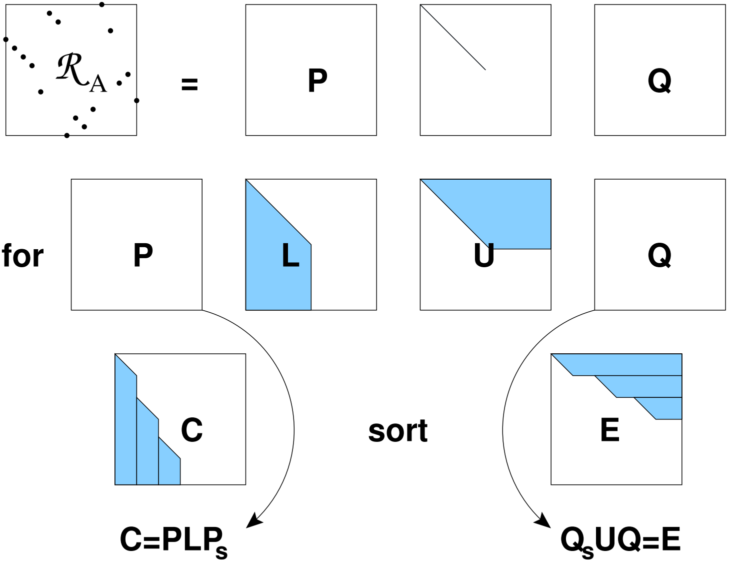
PLU

If A has generic ColRP

- $PLU(A)$ with RevLex order and row rot. $\rightsquigarrow P \begin{matrix} I_r & \\ & 0 \end{matrix} = \mathcal{R}_A$

In particular, if A has full column rank and $m = n$: $\rightsquigarrow P = \mathcal{R}_A$

Echelon forms



Small rank

When $r \ll m, n$, $O(mnr^{\omega-2})$ can be too expensive.
(Compressed sensing applications)

[Cheung Kwok Lau'12]: Compute the rank r and r linearly independent rows in $\tilde{O}(r^\omega + mn)$ probabilistic

Small rank

When $r \ll m, n$, $O(mnr^{\omega-2})$ can be too expensive.
(Compressed sensing applications)

[Cheung Kwok Lau'12]: Compute the rank r and r linearly independent rows in $\tilde{O}(r^{\omega} + mn)$ probabilistic

[Storjohann Yang'14:] Rank profile in $\tilde{O}(r^3 + mn)$ probabilistic.

Small rank

When $r \ll m, n$, $O(mnr^{\omega-2})$ can be too expensive.
(Compressed sensing applications)

[Cheung Kwok Lau'12]: Compute the rank r and r linearly independent rows in $\tilde{O}(r^\omega + mn)$ probabilistic

[Storjohann Yang'14:] Rank profile in $\tilde{O}(r^3 + mn)$ probabilistic.

[Storjohann Yang'15:] Rank profile in $\tilde{O}(r^\omega + mn)$ probabilistic.

Small rank

When $r \ll m, n$, $O(mnr^{\omega-2})$ can be too expensive.
(Compressed sensing applications)

[Cheung Kwok Lau'12]: Compute the rank r and r linearly independent rows in $\tilde{O}(r^\omega + mn)$ probabilistic

[Storjohann Yang'14:] Rank profile in $\tilde{O}(r^3 + mn)$ probabilistic.

[Storjohann Yang'15:] Rank profile in $\tilde{O}(r^\omega + mn)$ probabilistic.

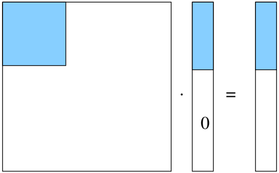
Can the rank profile matrix be computed in similar complexities?

[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.



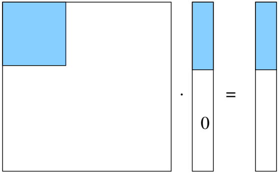
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$



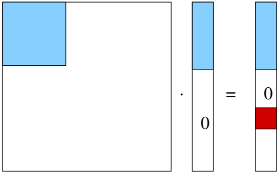
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$



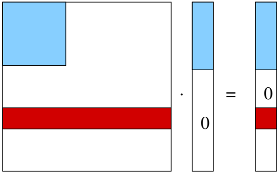
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3+mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$



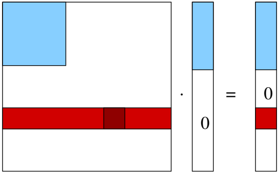
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$



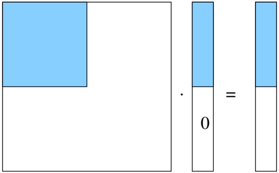
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$
2. Compute A_{s+1}^{-1} by rank 1 updates $\rightsquigarrow O(s^2)$



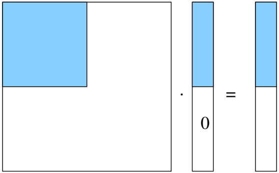
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(sn)$
2. Compute A_{s+1}^{-1} by rank 1 updates $\rightsquigarrow O(s^2)$



- Use the vector b to compress row linear dependency information

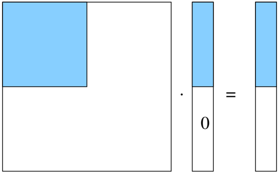
[Storjohann Yang'14] Linear System Oracle

Sketch of the $O\tilde{}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. $\rightsquigarrow O(s \log n)$
2. Compute A_{s+1}^{-1} by rank 1 updates $\rightsquigarrow O(s^2)$



- Use the vector b to compress row linear dependency information
- Improved by linear independence oracles

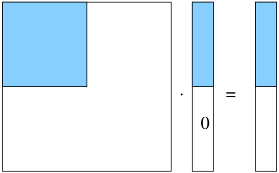
[Storjohann Yang'14] Linear System Oracle

Sketch of the $\tilde{O}(r^3 + mn)$ algorithm

Incrementally for $s = 1..\text{rank}(A)$, maintain

- an $s \times s$ invertible sub-matrix A_s of A.
- its inverse A_s^{-1}
- a partial solution $A_s x_s = b_s$ to a linear system $Ax = b$.

1. Use A_s^{-1} to find the next row and column to append to A_s. ⇝ $O(s \log n)$
2. Compute A_{s+1}^{-1} by rank 1 updates ⇝ $O(s^2)$



- Use the vector b to compress row linear dependency information
- Improved by linear independence oracles

Lexico. search with rotations ⇝ computes $\mathcal{R}_A$

[Storjohann Yang'15] Relaxed matrix inverse

Sketch of the algorithm: RowRP in $O\tilde{}(r^\omega + mn)$

1. Insted of building A_s^{-1} iteratively ($O(r^3)$), use an asymptotically fast relaxation scheme $O(r^\omega)$.
2. Requires to deal with only r columns in generic column RP.
3. Ensured by a call to [Cheung Kwok Lau'12] $+$ Toeplitz preconditionner
4. Returns the row rank profile

[Storjohann Yang'15] Relaxed matrix inverse

Sketch of the algorithm: RowRP in $\tilde{O}(r^\omega + mn)$

1. Insted of building A_s^{-1} iteratively ($O(r^3)$), use an asymptotically fast relaxation scheme $O(r^\omega)$.
2. Requires to deal with only r columns in generic column RP.
3. Ensured by a call to [Cheung Kwok Lau'12] + Toeplitz preconditionner
4. Returns the row rank profile

Problem: step 3 loses information required for the $\mathcal{R}_A$.

[Storjohann Yang'15] Relaxed matrix inverse

Sketch of the algorithm: RowRP in $\tilde{O}(r^\omega + mn)$

1. Insted of building A_s^{-1} iteratively ($O(r^3)$), use an asymptotically fast relaxation scheme $O(r^\omega)$.
2. Requires to deal with only r columns in generic column RP.
3. Ensured by a call to [Cheung Kwok Lau'12] + Toeplitz preconditionner
4. Returns the row rank profile

Problem: step 3 loses information required for the $\mathcal{R}_A$.

Solution for $\mathcal{R}_A$ in $\tilde{O}(r^\omega + mn)$

1. Compute the RowRP $\mathcal{I}$ by [Storjohann Yang'15] on A
2. Compute the ColRP $\mathcal{J}$ by [Storjohann Yang'15] on A^T

[Storjohann Yang'15] Relaxed matrix inverse

Sketch of the algorithm: RowRP in $\tilde{O}(r^\omega + mn)$

1. Insted of building A_s^{-1} iteratively ($O(r^3)$), use an asymptotically fast relaxation scheme $O(r^\omega)$.
2. Requires to deal with only r columns in generic column RP.
3. Ensured by a call to [Cheung Kwok Lau'12] $+$ Toeplitz preconditionner
4. Returns the row rank profile

Problem: step 3 loses information required for the $\mathcal{R}_A$.

Solution for $\mathcal{R}_A$ in $\tilde{O}(r^\omega + mn)$

1. Compute the RowRP $\mathcal{I}$ by [Storjohann Yang'15] on A
2. Compute the ColRP $\mathcal{J}$ by [Storjohann Yang'15] on A^T
3. Extract the $r \times r$ submatrix $A_r = A_{\mathcal{I},\mathcal{J}}$
4. Compute the LUP decomp of A_r with col. rotations

[Storjohann Yang'15] Relaxed matrix inverse

Sketch of the algorithm: RowRP in $\tilde{O}(r^\omega + mn)$

1. Insted of building A_s^{-1} iteratively ($O(r^3)$), use an asymptotically fast relaxation scheme $O(r^\omega)$.
2. Requires to deal with only r columns in generic column RP.
3. Ensured by a call to [Cheung Kwok Lau'12] + Toeplitz preconditionner
4. Returns the row rank profile

Problem: step 3 loses information required for the $\mathcal{R}_A$.

Solution for $\mathcal{R}_A$ in $\tilde{O}(r^\omega + mn)$

1. Compute the RowRP $\mathcal{I}$ by [Storjohann Yang'15] on A
2. Compute the ColRP $\mathcal{J}$ by [Storjohann Yang'15] on A^T
3. Extract the $r \times r$ submatrix $A_r = A_{\mathcal{I},\mathcal{J}}$
4. Compute the LUP decomp of A_r with col. rotations
5. Recover $\mathcal{R}_A$ by inflating $\mathcal{R}_{A_r} = P$ with zeroes.

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**
  $\rightsquigarrow$ harnesses floating point efficiency
  $\rightsquigarrow$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  $\rightsquigarrow$ **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**
  $\rightsquigarrow$ harnesses floating point efficiency
  $\rightsquigarrow$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  $\rightsquigarrow$ **architecture oblivious vs aware** algorithms [Gustavson 07]
- New pivoting strategies revealing **all rank profile informations**
  $\rightsquigarrow$ **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**
  $\rightsquigarrow$ harnesses floating point efficiency
  $\rightsquigarrow$ embarrassingly easy parallelization (and fault tolerance)
- Favor **tiled recursive** algorithms
  $\rightsquigarrow$ **architecture oblivious vs aware** algorithms [Gustavson 07]
- New pivoting strategies revealing **all rank profile informations**
  $\rightsquigarrow$ **tournament pivoting**? [Demmel, Grigori and Xiang 11]

Thank you