High performance computer algebra: how to use floating point arithmetic exactly?

Journée Informatique et Calcul

Clément Pernet

LIP, Univ. Grenoble Alpes

29 Septembre 2015

Exact linear algebra

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: $\mathsf{K}[X]$ for K any of the above

Exact linear algebra

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: $\mathsf{K}[X]$ for K any of the above

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

Exact linear algebra

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: $\mathsf{K}[X]$ for K any of the above

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

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

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

Exact linear algebra

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: $\mathsf{K}[X]$ for K any of the above

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

Matrices can be

Dense: store all coefficients

Sparse: store the non-zero coefficients only

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

Need for genericity.

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.

Outline

1. Choosing the underlying arithmetic
   - Using machine word arithmetic
   - Larger field sizes
2. Algorithmic: reductions and building blocks
3. Parallel exact linear algebra

Outline

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

Available CPU arithmetic

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

Which computer arithmetic ?

Many base fields/rings to support

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

Available CPU arithmetic

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

Dense linear algebra over $\mathbb{Z}_p$ 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)	$(\frac{1+3^\ell}{2})^2 \lfloor \frac{n}{2^\ell} \rfloor (p-1)^2$	$9^\ell \lfloor \frac{n}{2^\ell} \rfloor \left(\frac{p-1}{2}\right)^2$

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

Delayed modular reductions

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

When to reduce ?

Bound the values of all intermediate computations.

- A priori:

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

- Maintain locally a bounding interval on all matrices computed

Computing over fixed size integers

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

1. use CPU's **integer arithmetic units**
   `y += 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
mulsd   -56(%rbp), %xmm0
addsd   %xmm0, %xmm1
movq    %xmm1, %rax
```

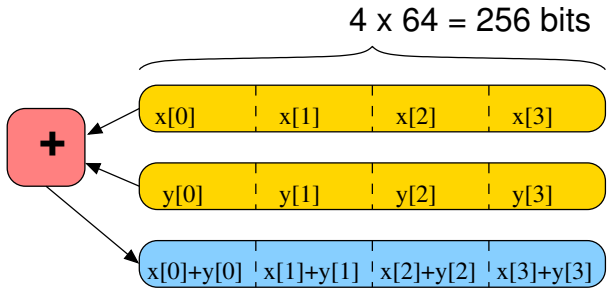
```
vinsertf128   $0x1, 16(%rcx,%rax), %ymm0,
vmulpd        %ymm1, %ymm0, %ymm0
vaddpd        (%rsi,%rax),%ymm0, %ymm0
vmovapd       %ymm0, (%rsi,%rax)
```

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

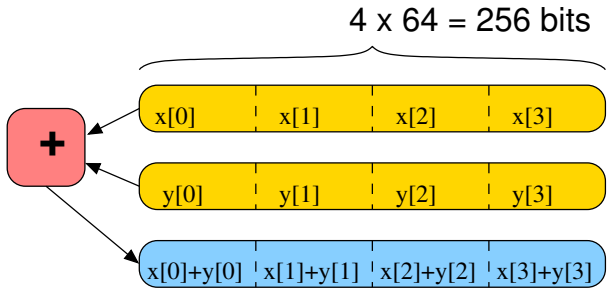


Exploiting *in-core* parallelism

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

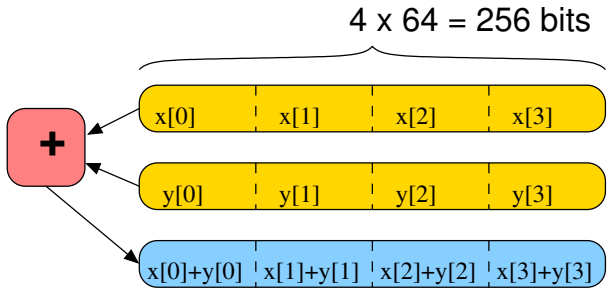
throughput of 1 op/ Cycle for all arithmetic ops considered here

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



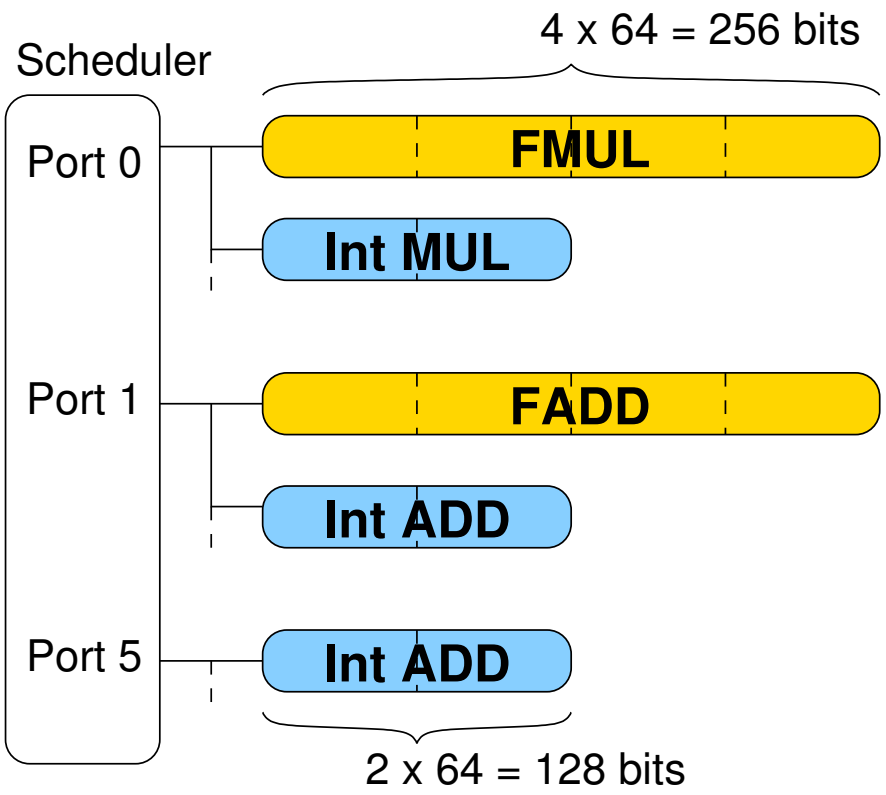
Pipeline: amortize the latency of an operation when used repeatedly

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:

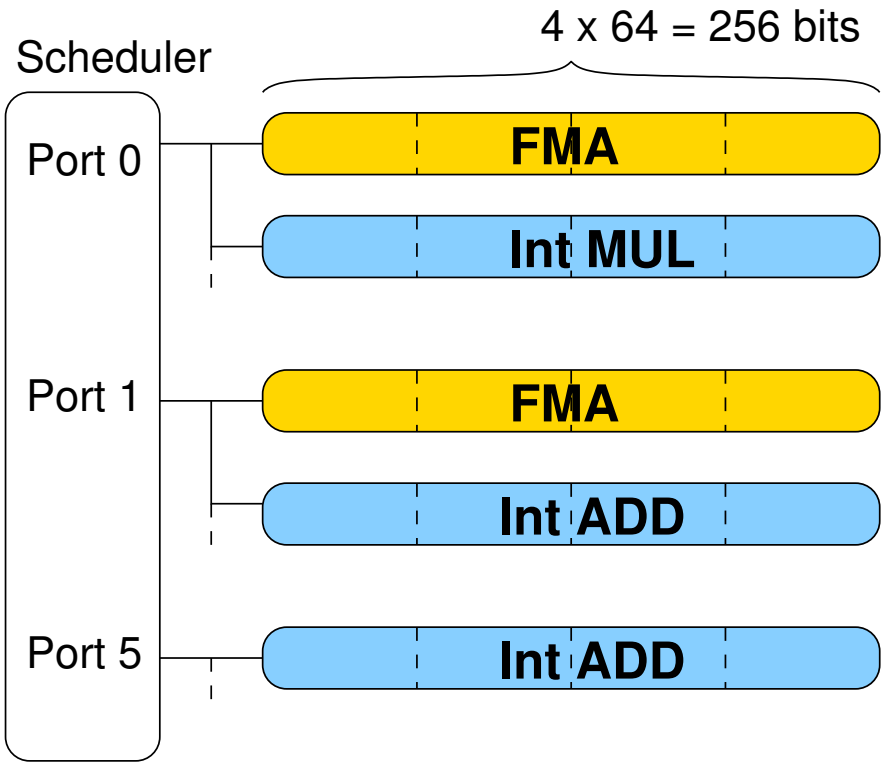
- 1 Floating Pt. Mul $\times$ 4
- 1 Floating Pt. Add $\times$ 4

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

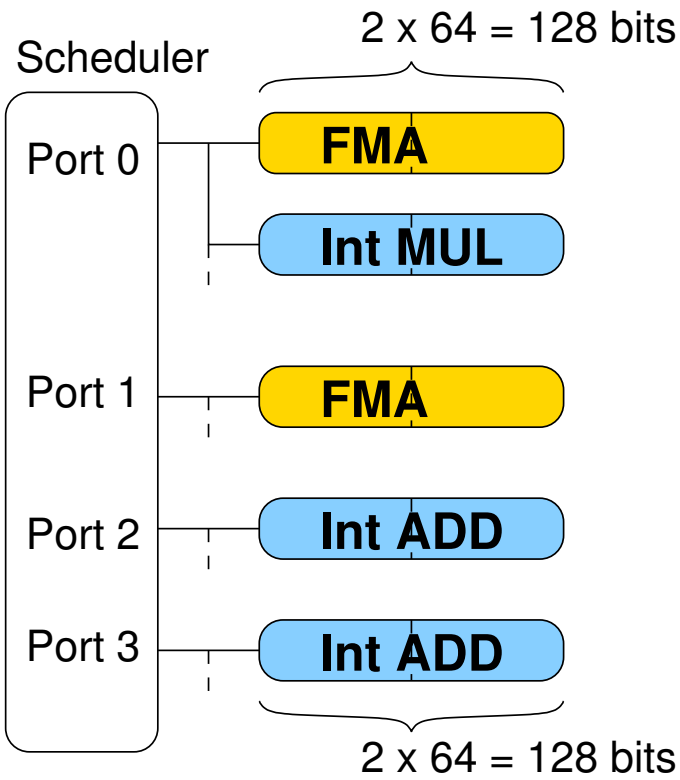
Or:

- 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

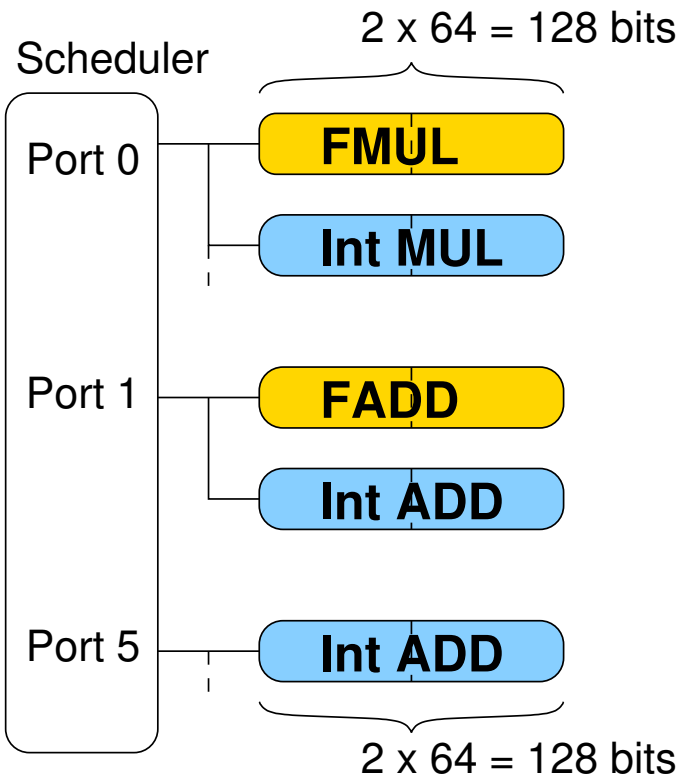
Or:

- 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*	*# 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 $\times$ (# 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								
AVX1	FP								
AMD Bulldozer	INT								
FMA4	FP								
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**	
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 $\times$ (# 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								
FMA4	FP								
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**	
FMA4	FP	128			2	4	2.1	**16.8**	
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								
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 $\times$ (# 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								
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 $\times$ (# 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**	**8.93**

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

Computing over fixed size integers: ressources

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

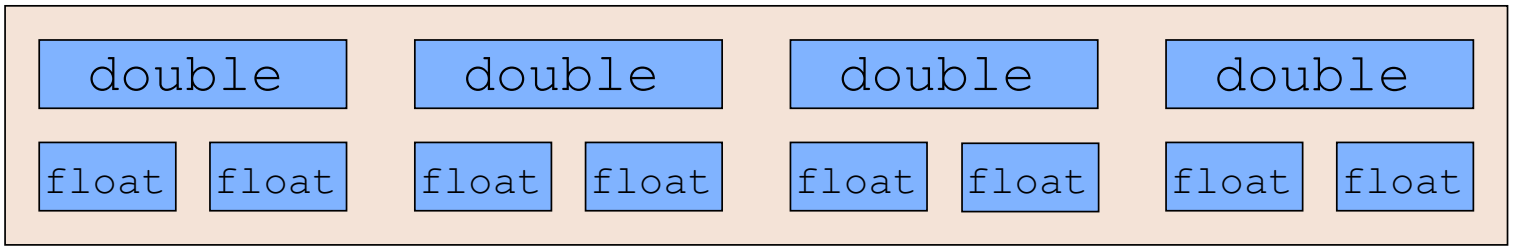
Experiments:

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

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

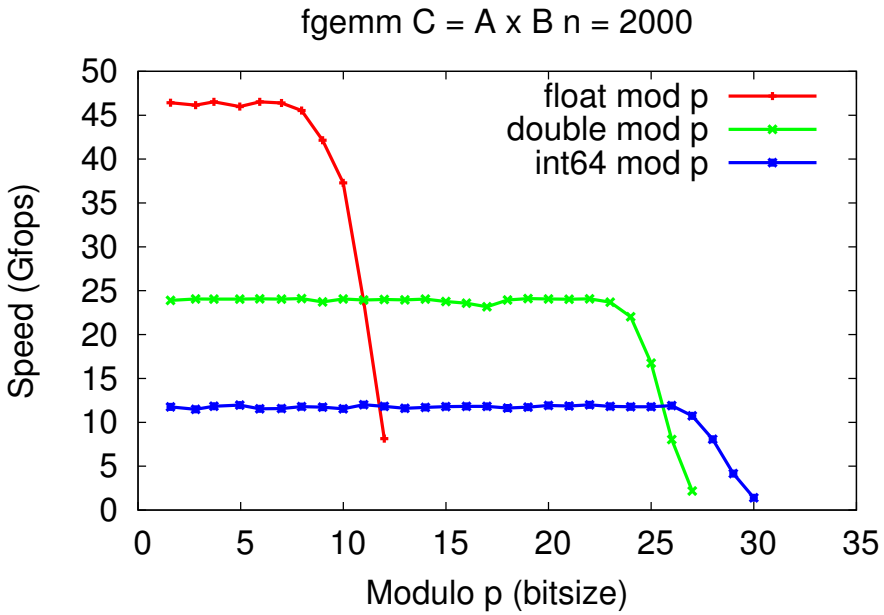
Integer Packing

32 bits: half the precision twice the speed



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

Computing over fixed size integers

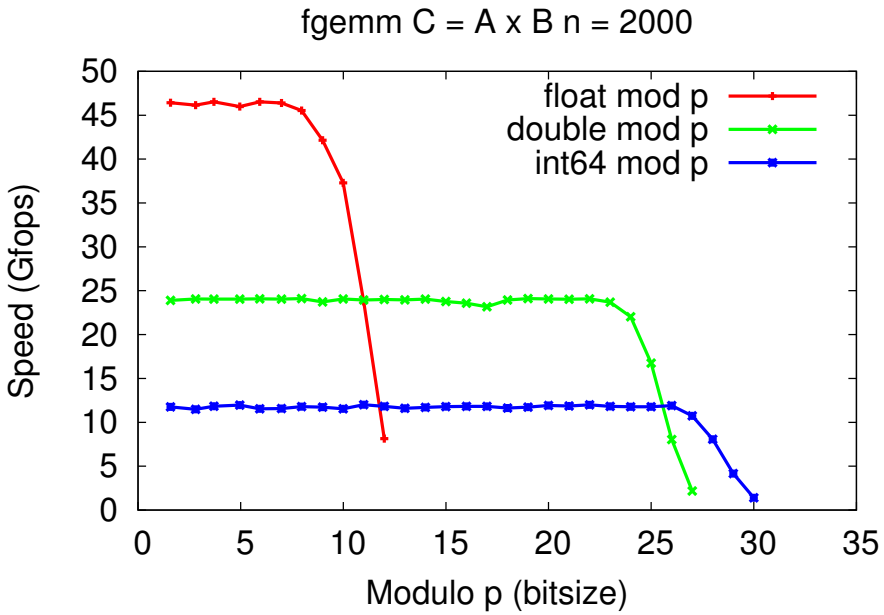


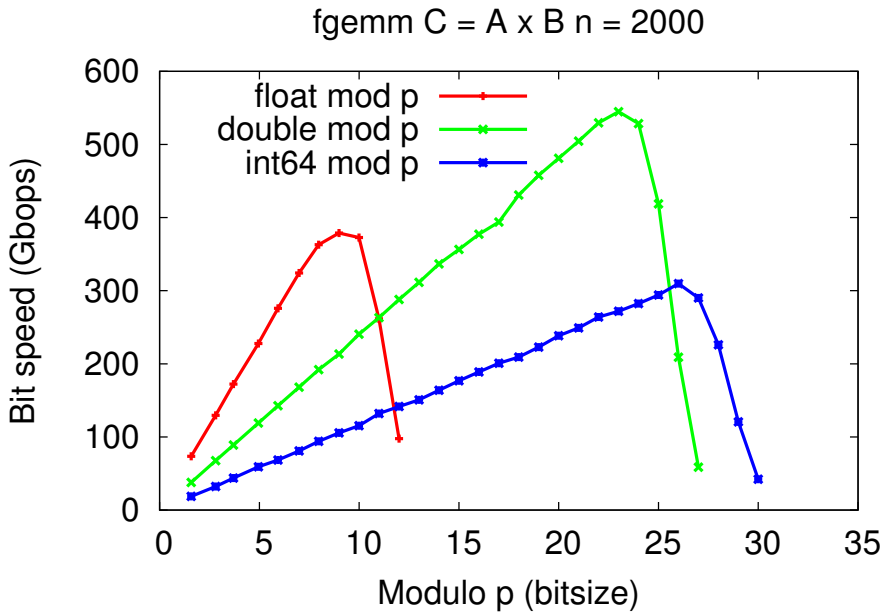
SandyBridge i5-3320M@3.3Ghz. $n = 2000$.

Take home message

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

Computing over fixed size integers





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

Take home message

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

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

As before:

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

Which integer arithmetic?

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

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

As before:

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

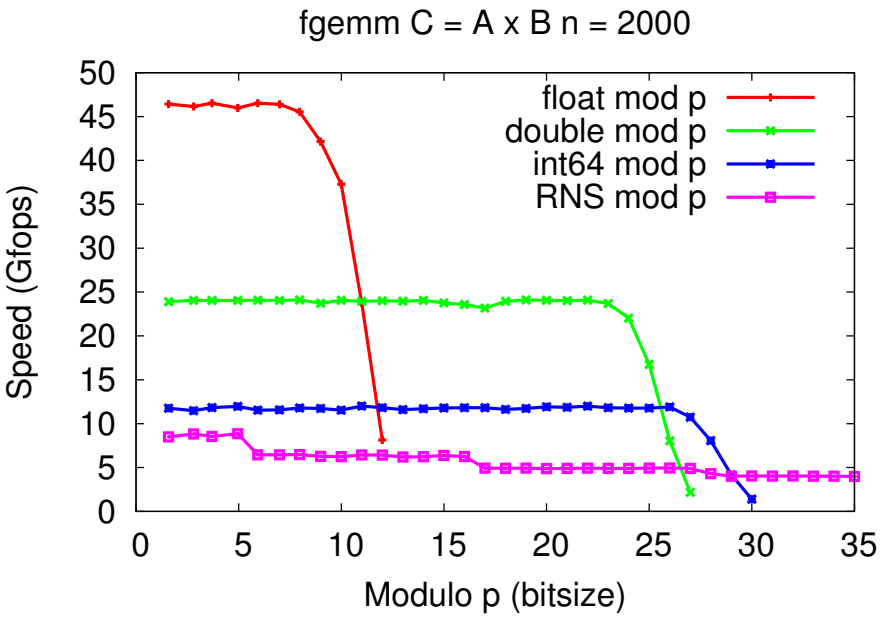
Which integer arithmetic?

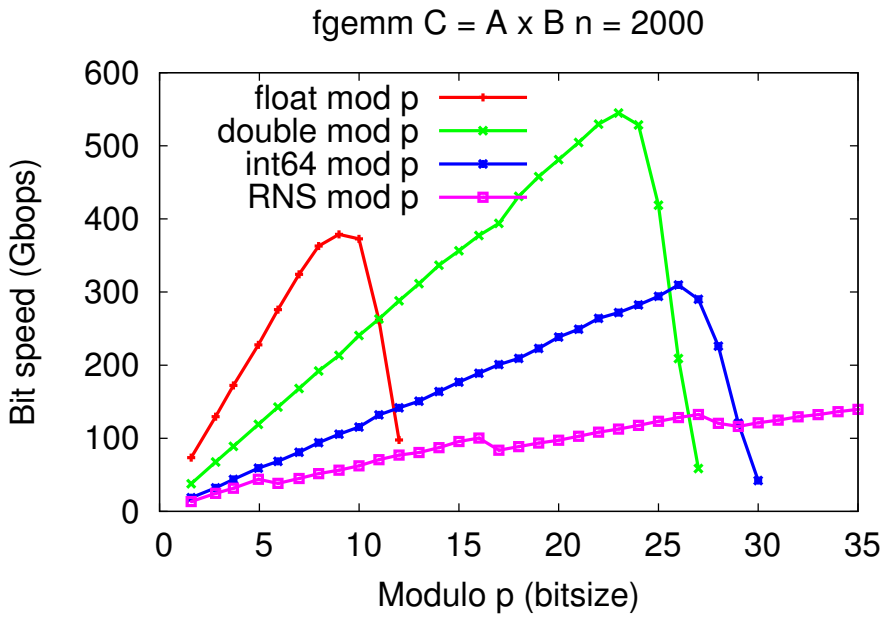
1. big integers (GMP)
2. fixed size multiprecision (Givaro-RecInt)
3. Residue Number Systems (Chinese Remainder theorem)
   $\rightsquigarrow$ using moduli delivering optimum bitspeed

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

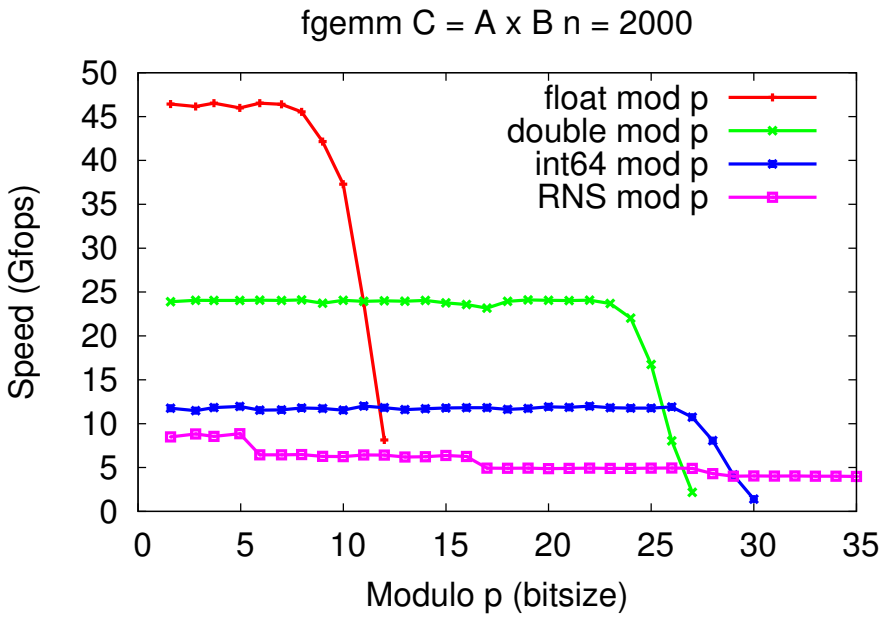
$n = 1000$, on an Intel SandyBridge.

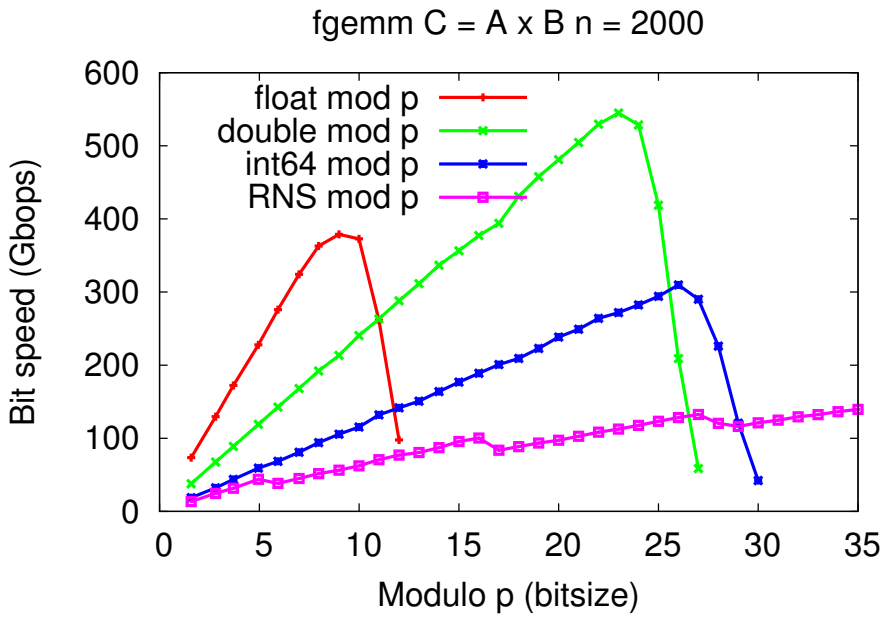
In practice

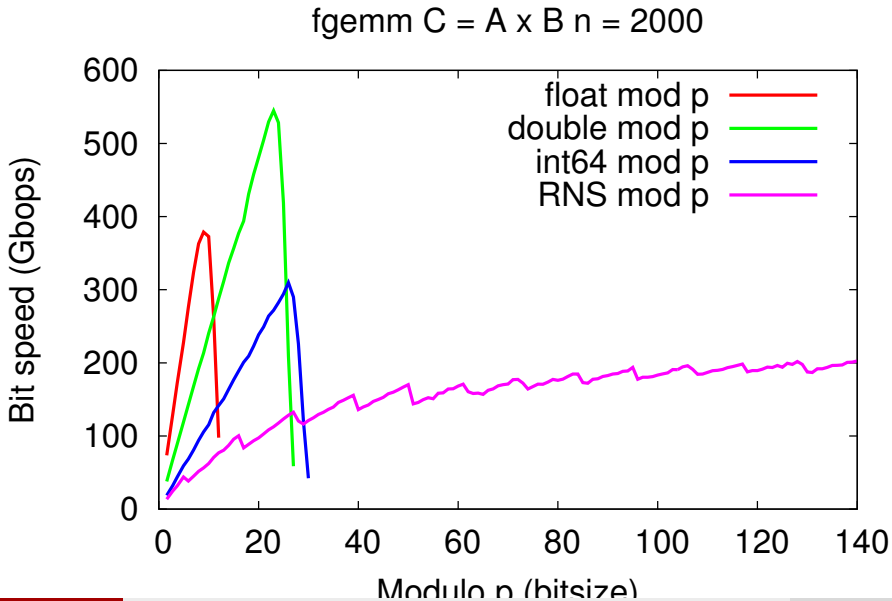

fgemm C = A x B n = 2000
Speed (Gfops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p
0
5
10
15
20
25
30
35
40
45
50


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
5
10
15
20
25
30
35

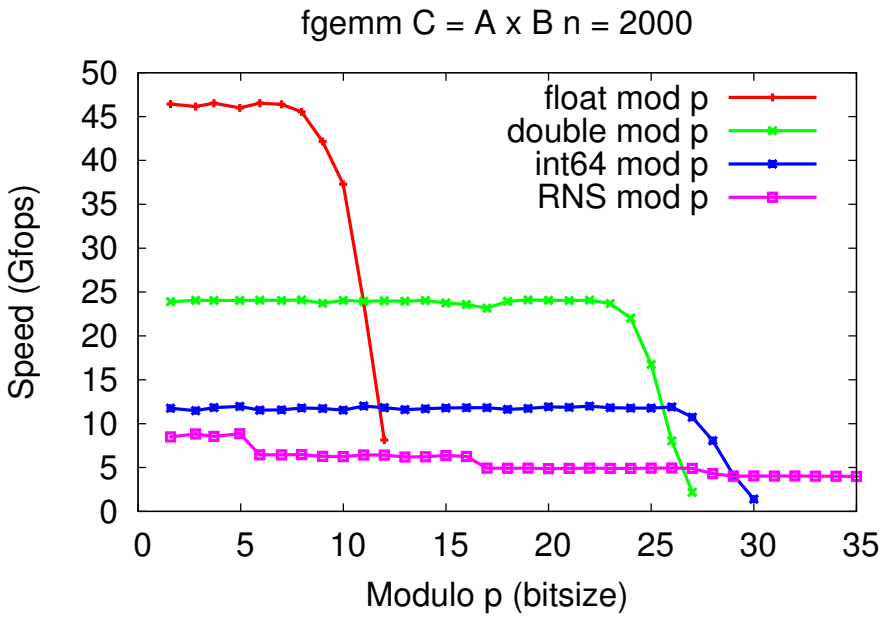
In practice

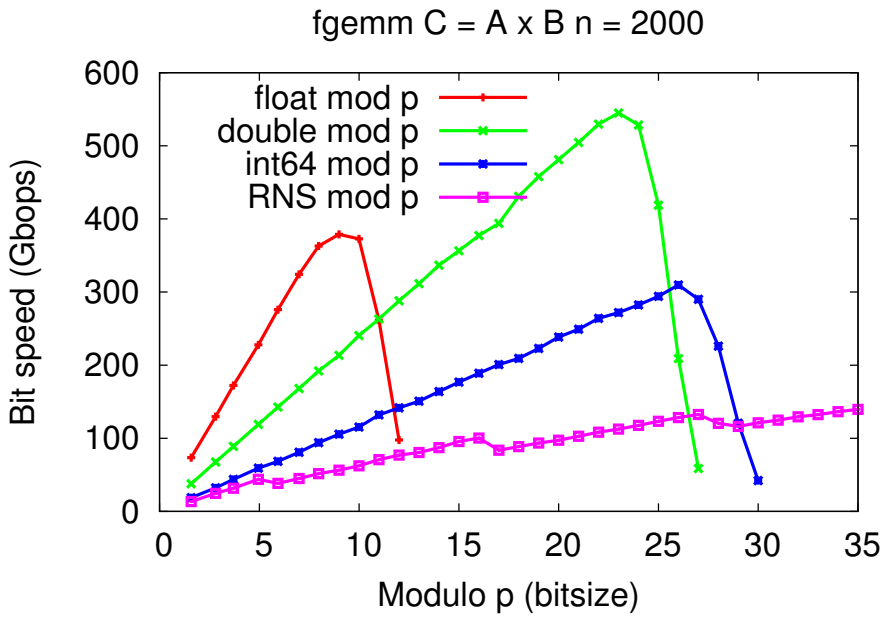

fgemm C = A x B n = 2000
Speed (Gfops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p
0
5
10
15
20
25
30
35
40
45
50

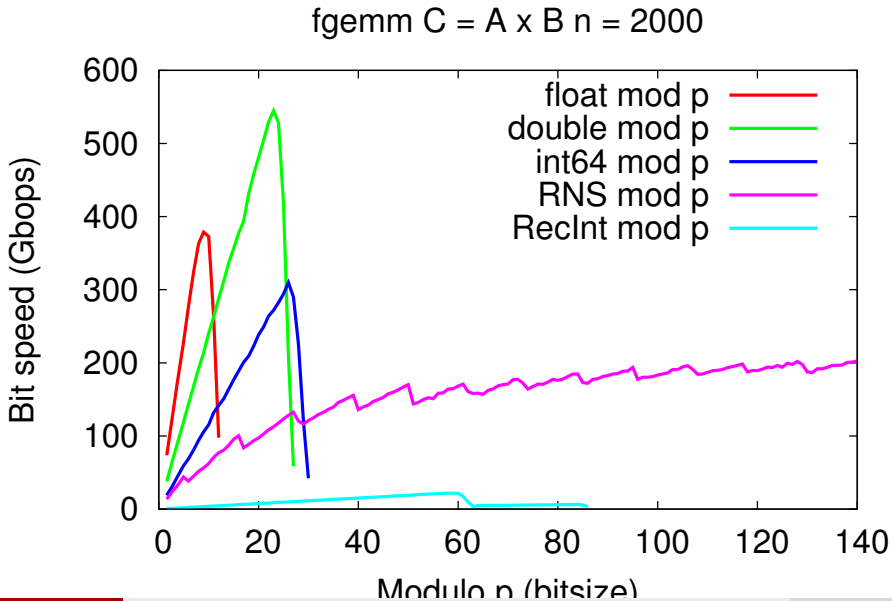

fgemm C = A x B n = 2000
Bit speed (Gbops)
Modulo p (bitsize)
float mod p
double mod p
int64 mod p
RNS mod p
0
100
200
300
400
500
600
5
10
15
20
25
30
35


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
20
40
60
80
100
120
140

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


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
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
20
40
60
80
100
120
140

Outline

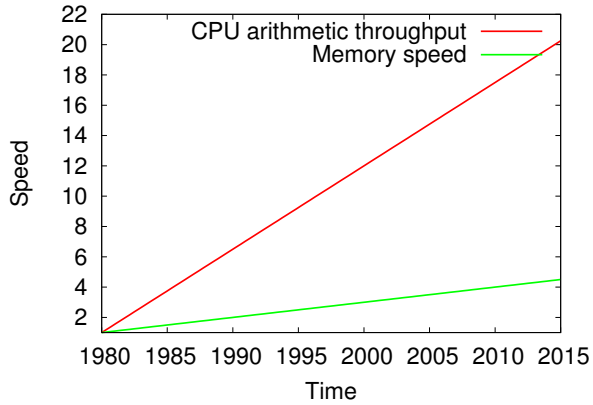
Reductions to building blocks

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

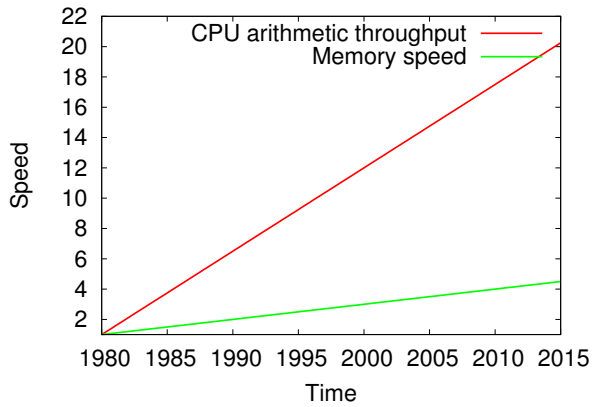
Memory access pattern

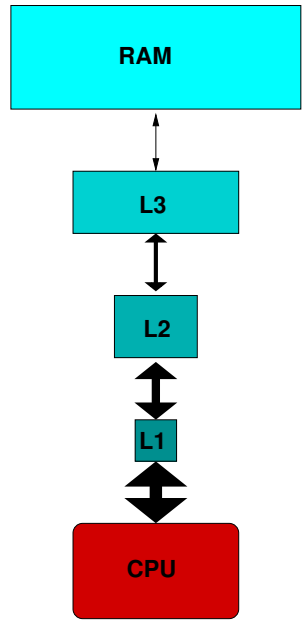
- **The memory wall:** communication speed improves slower than arithmetic



Memory access pattern

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

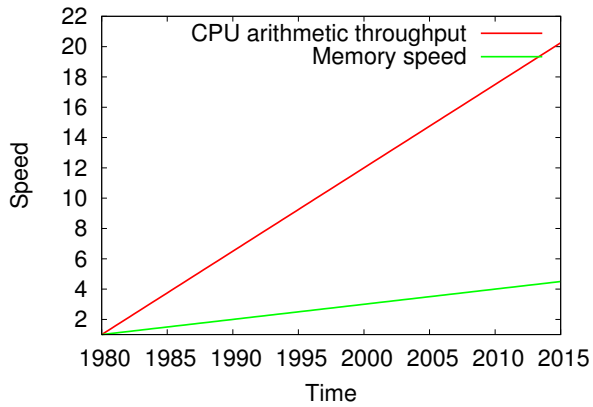


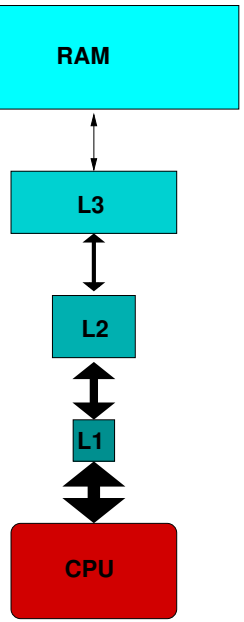


Memory access pattern

- **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, Danilevskǐi, 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 $\rightsquigarrow O(n^\omega)$

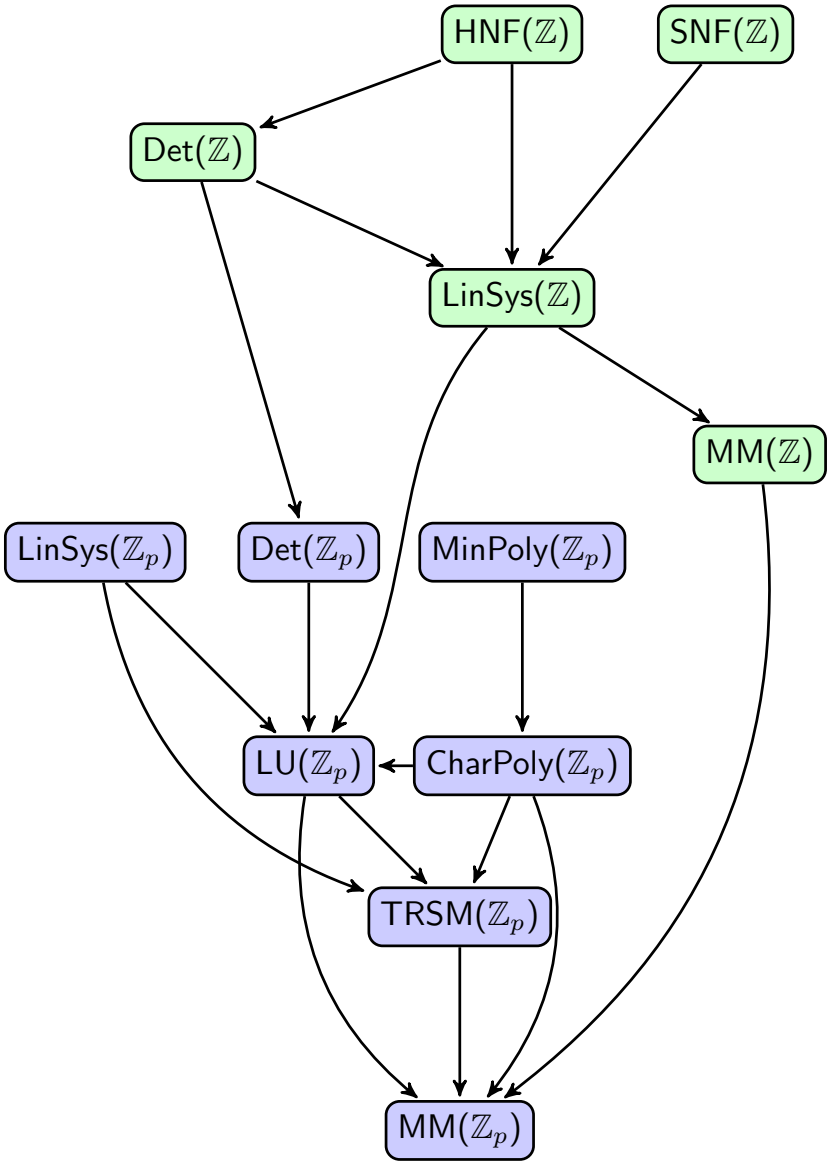
[Strassen 69]:	$O(n^{2.807})$
⋮	
[Schönhage 81]	$O(n^{2.52})$
⋮	
[Coppersmith, Winograd 90]	$O(n^{2.375})$
⋮	
[Le Gall 14]	$O(n^{2.3728639})$

Other operations

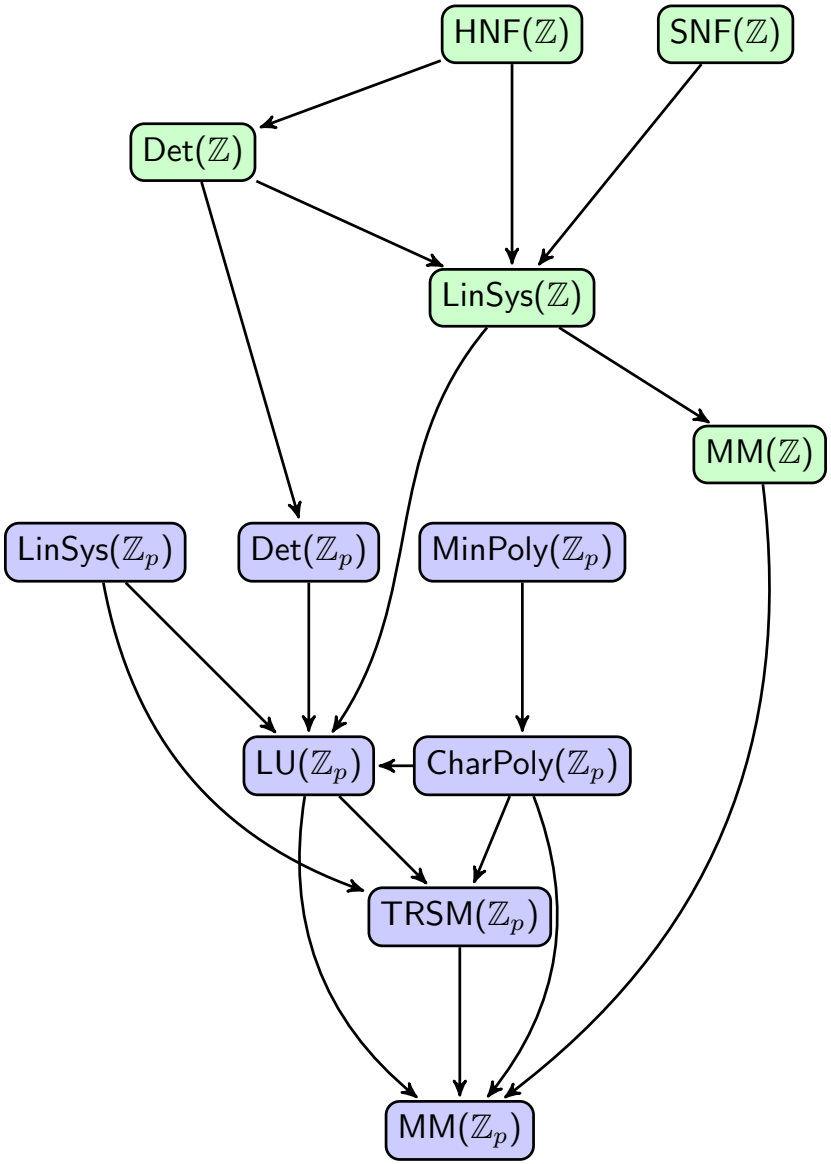
[Strassen 69]:	Inverse in $O(n^\omega)$
[Schönhage 72]:	QR in $O(n^\omega)$
[Bunch, Hopcroft 74]:	LU in $O(n^\omega)$
[Ibarra & *al.* 82]:	Rank in $O(n^\omega)$
[Keller-Gehrig 85]:	CharPoly in $O(n^\omega \log n)$

MatMul has become a building block in theoretical reductions

Reductions: theory



Reductions: theory

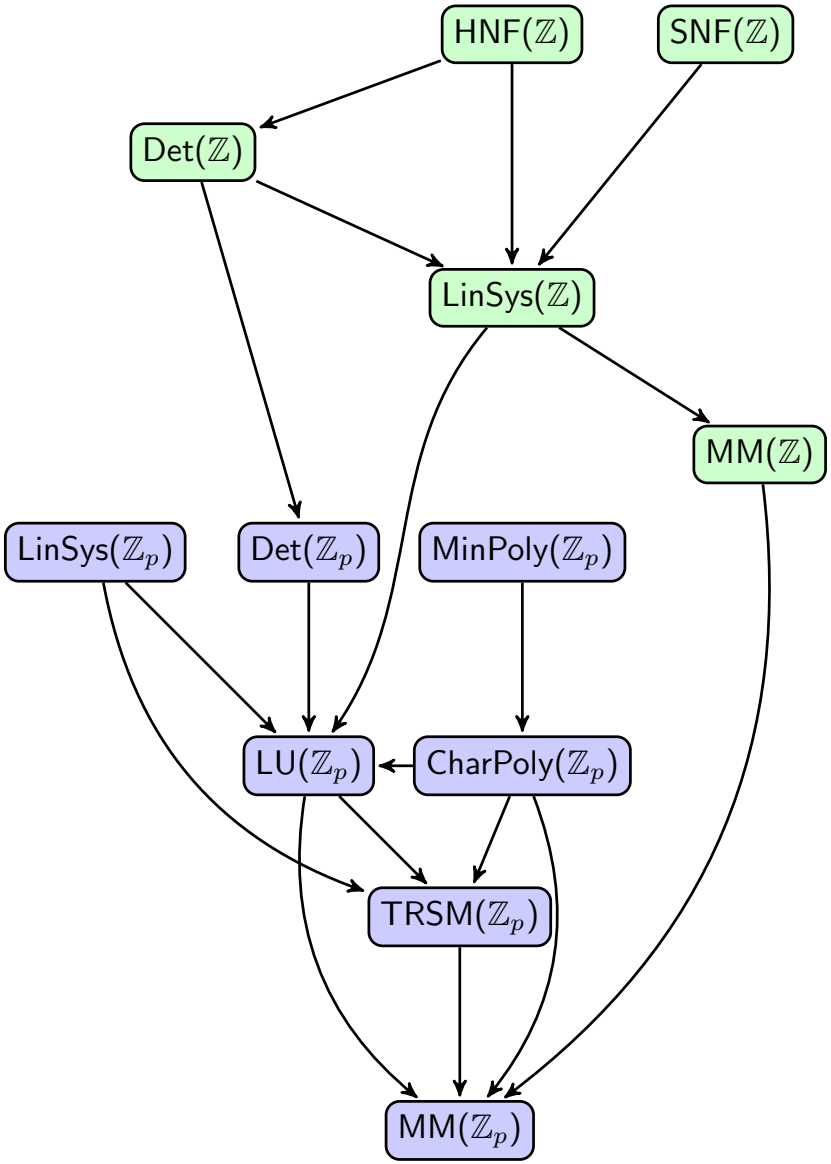


Common mistrust

Fast linear algebra is

- ✗ never faster
- ✗ numerically unstable

Reductions: theory and practice



Common mistrust

Fast linear algebra is

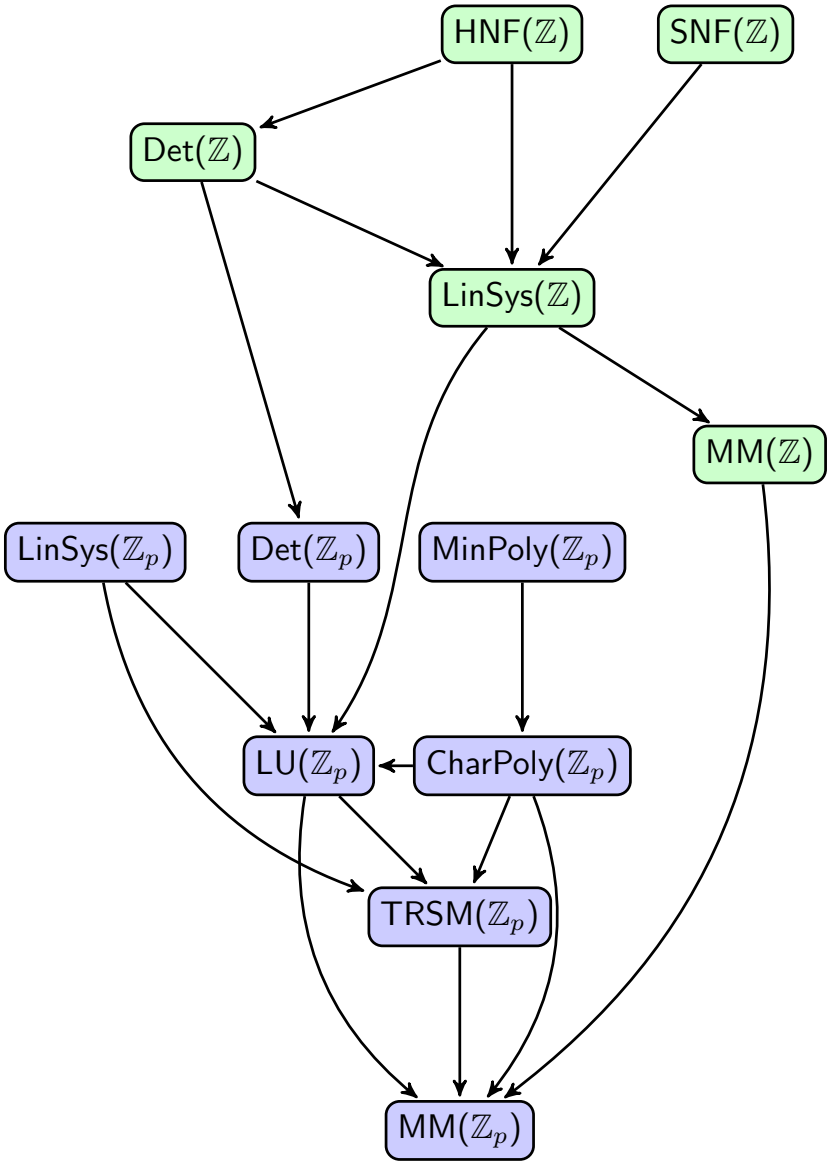
- ✗ never faster
- ✗ numerically unstable

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**

$\rightsquigarrow$ 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

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

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

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

- Fastest integer arithmetic: `double`
- Cache optimizations

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

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

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

Ingedients [FFLAS-FFPACK library]

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

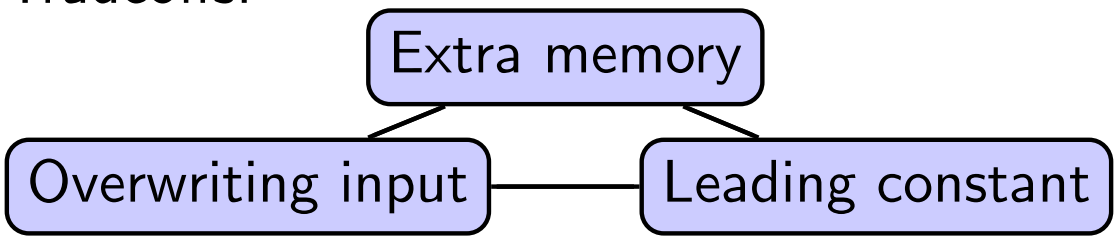
$$\rightsquigarrow \quad 9^\ell \left\lfloor \frac{k}{2^\ell} \right\rfloor \left(\frac{p-1}{2}\right)^2 < 2^{53}$$

- Fastest integer arithmetic: `double`
- Cache optimizations

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

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

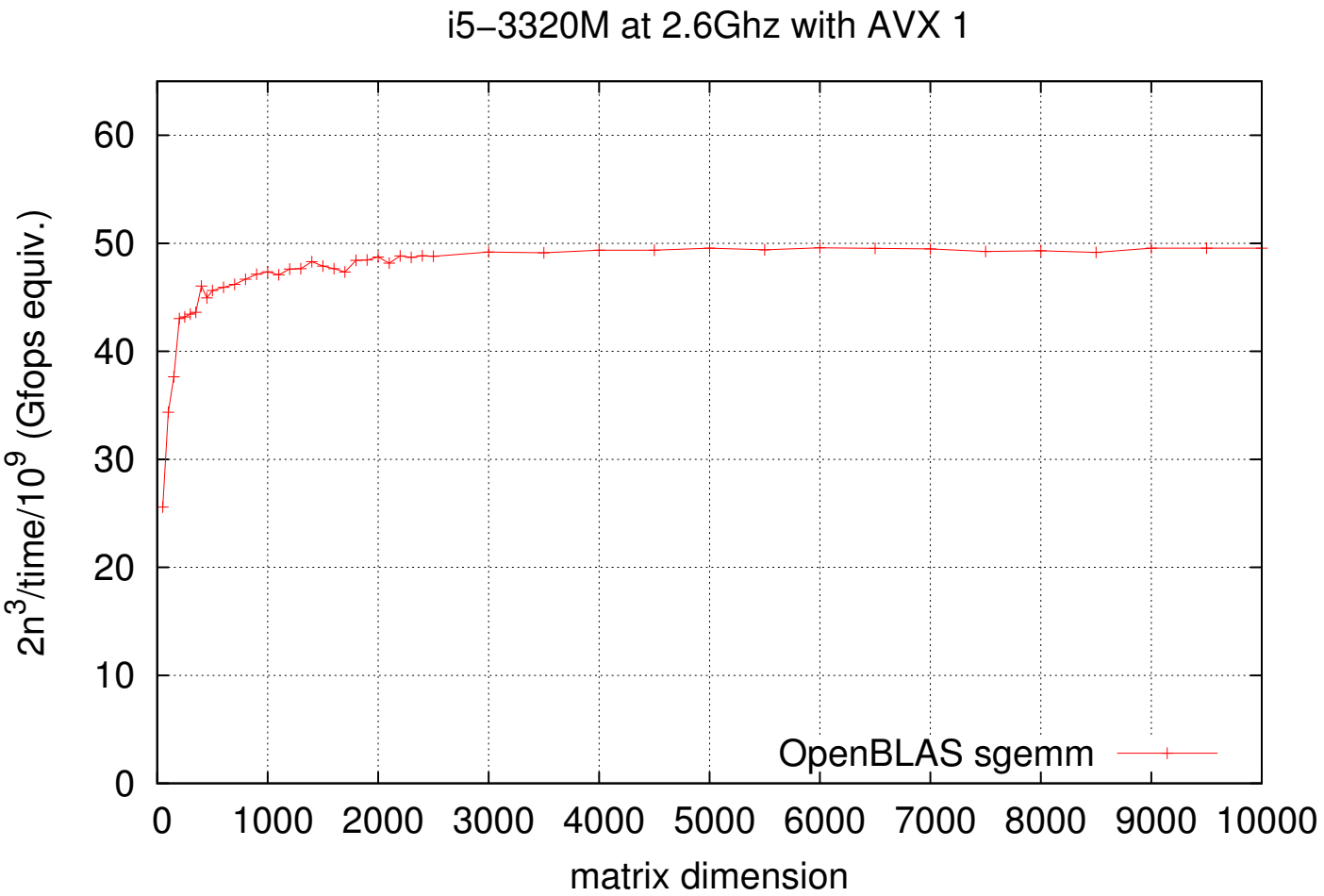
with memory efficient schedules [Boyer, Dumas, P. and Zhou 09]
Tradeoffs:



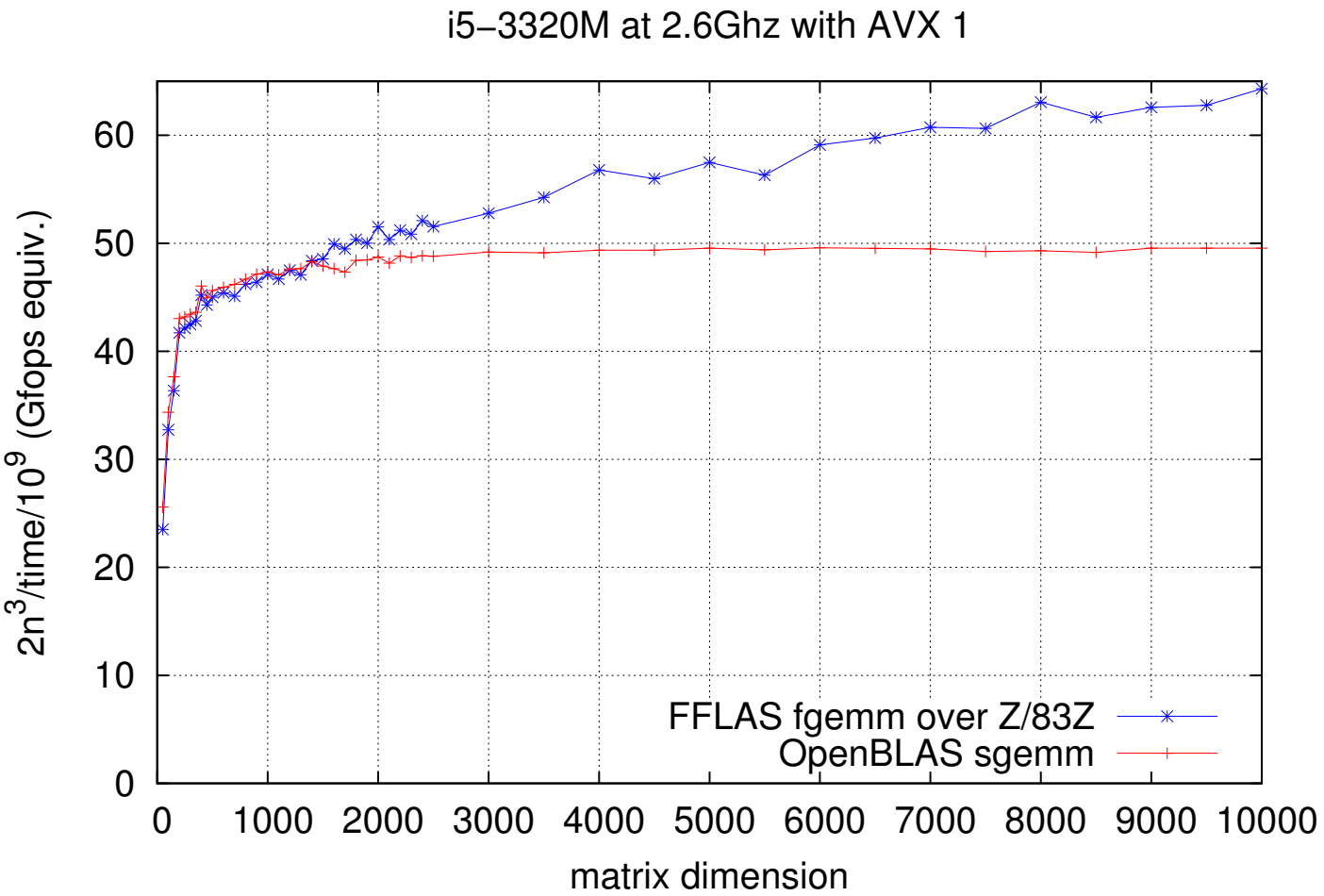
Fully in-place in

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

Sequential Matrix Multiplication

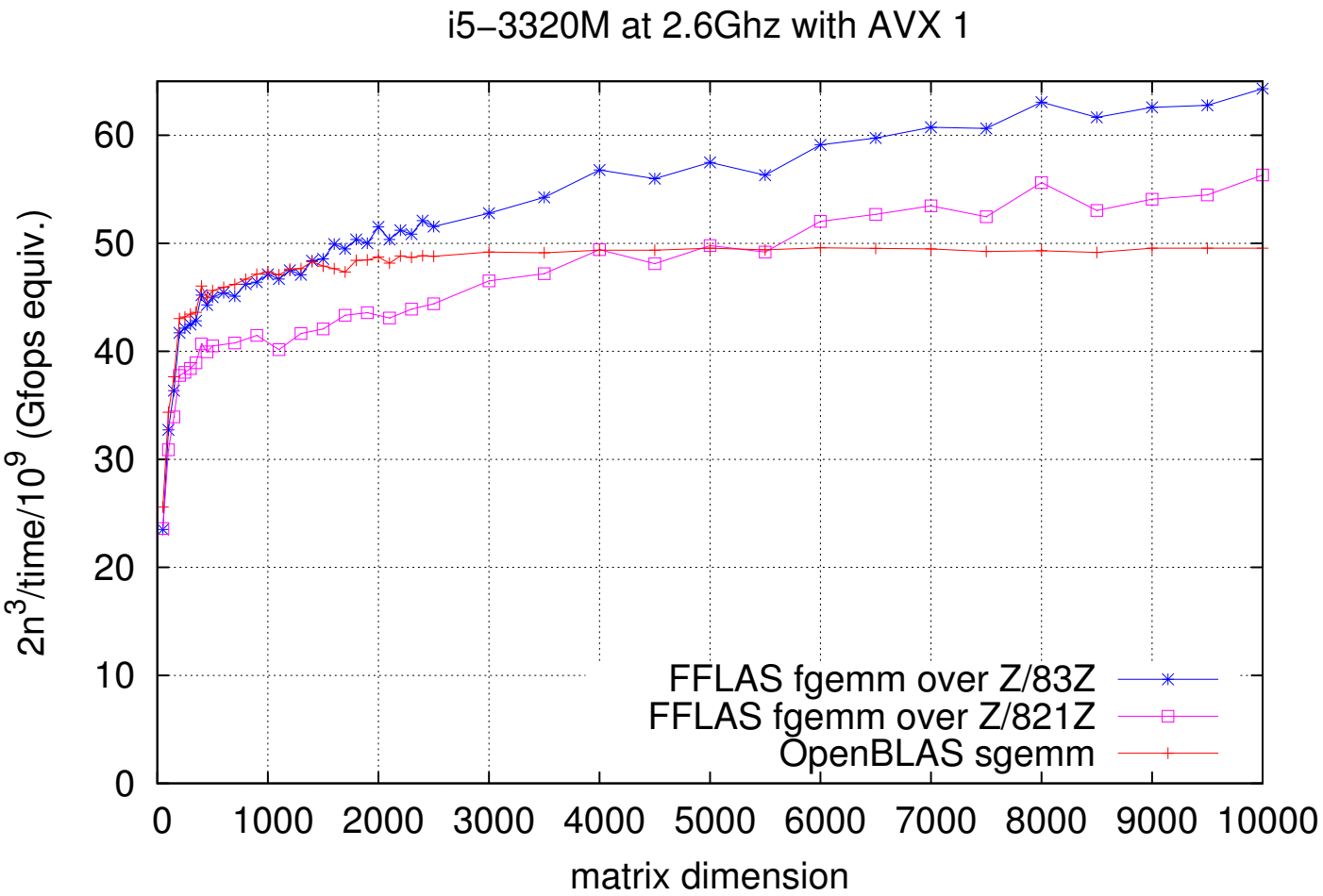


Sequential Matrix Multiplication



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

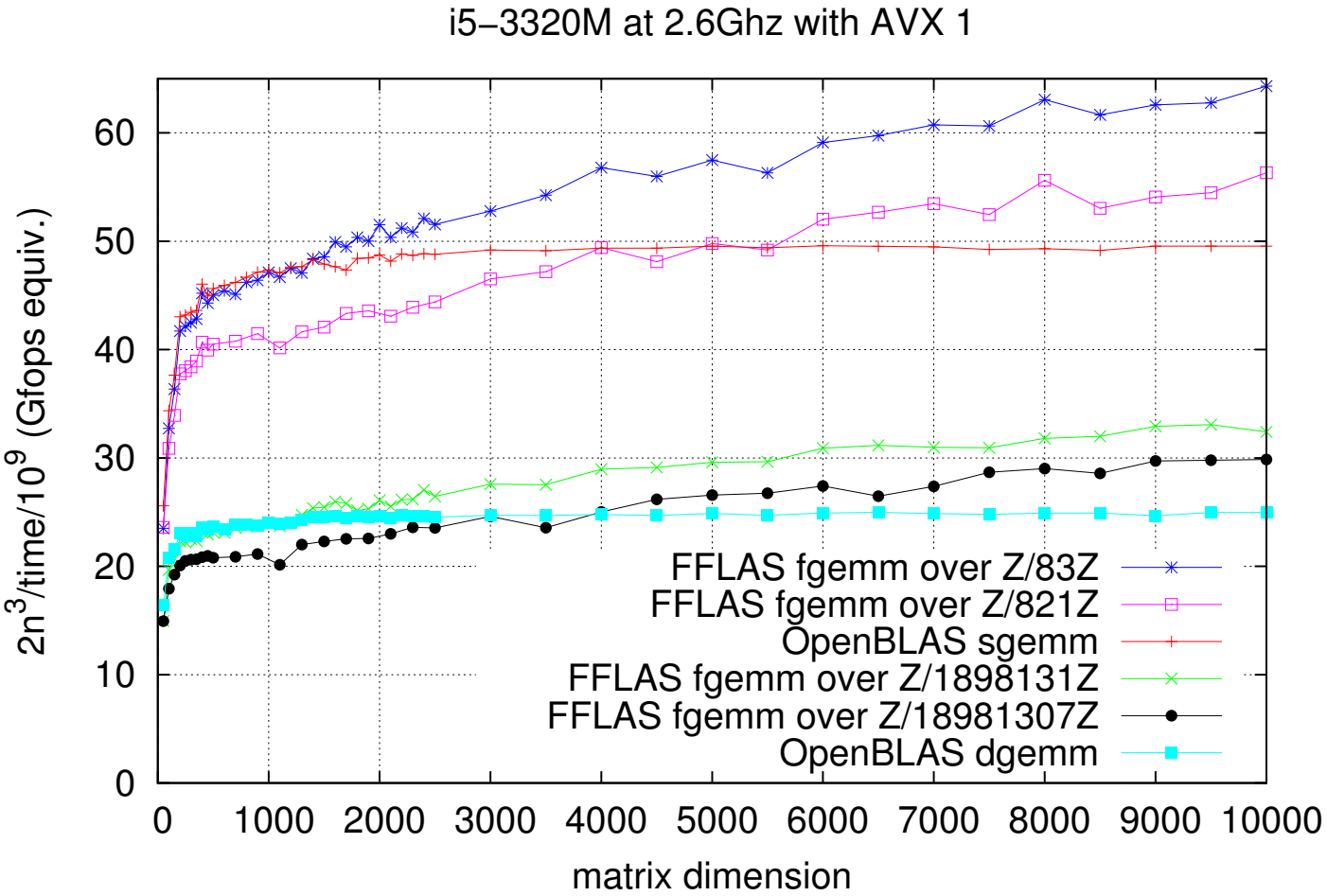
Sequential Matrix Multiplication



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

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

Sequential Matrix Multiplication



$p = 83$, $\rightsquigarrow$ 1 mod / 10000 mul.
$p = 821$, $\rightsquigarrow$ 1 mod / 100 mul.
$p = 1898131$, $\rightsquigarrow$ 1 mod / 10000 mul.
$p = 18981307$, $\rightsquigarrow$ 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

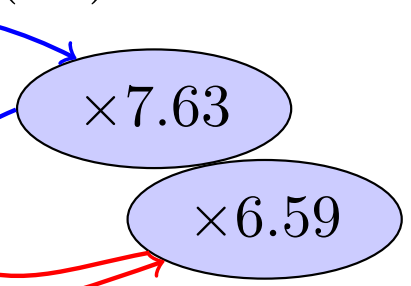
Reductions in dense linear algebra

LU decomposition

- Block recursive algorithm $\rightsquigarrow$ 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

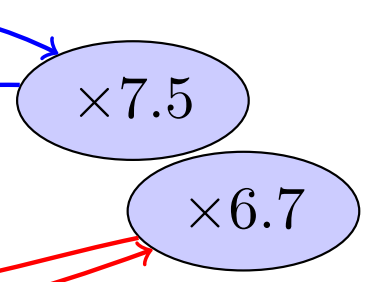


Characteristic Polynomial

- 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



Outline

Parallelization

Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$

  $\rightsquigarrow$ fine grain

  $\rightsquigarrow$ block iterative algorithms
- regular task load
- Numerical stability constraints

Parallelization

Parallel numerical linear algebra

- cost invariant wrt. splitting
  - $O(n^3)$

  $\rightsquigarrow$ fine grain

  $\rightsquigarrow$ block iterative algorithms
- regular task load
- Numerical stability constraints

Exact linear algebra specificities

- cost affected by the splitting
  - $O(n^\omega)$ for $\omega < 3$
  - modular reductions

  $\rightsquigarrow$ coarse grain

  $\rightsquigarrow$ recursive algorithms
- rank deficiencies

  $\rightsquigarrow$ unbalanced task loads

Ingredients for the parallelization

Criteria

- good performances
- portability across architectures
- abstraction for simplicity

Challenging key point: scheduling as a plugin

Program: only describes where the parallelism lies

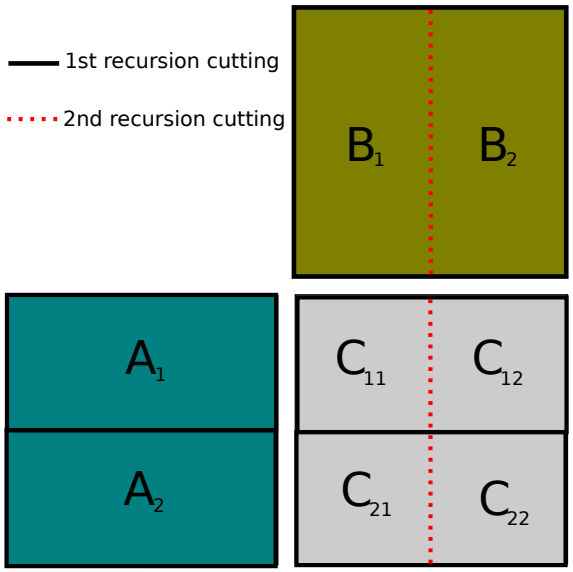
Runtime: scheduling & mapping, depending on the context of execution

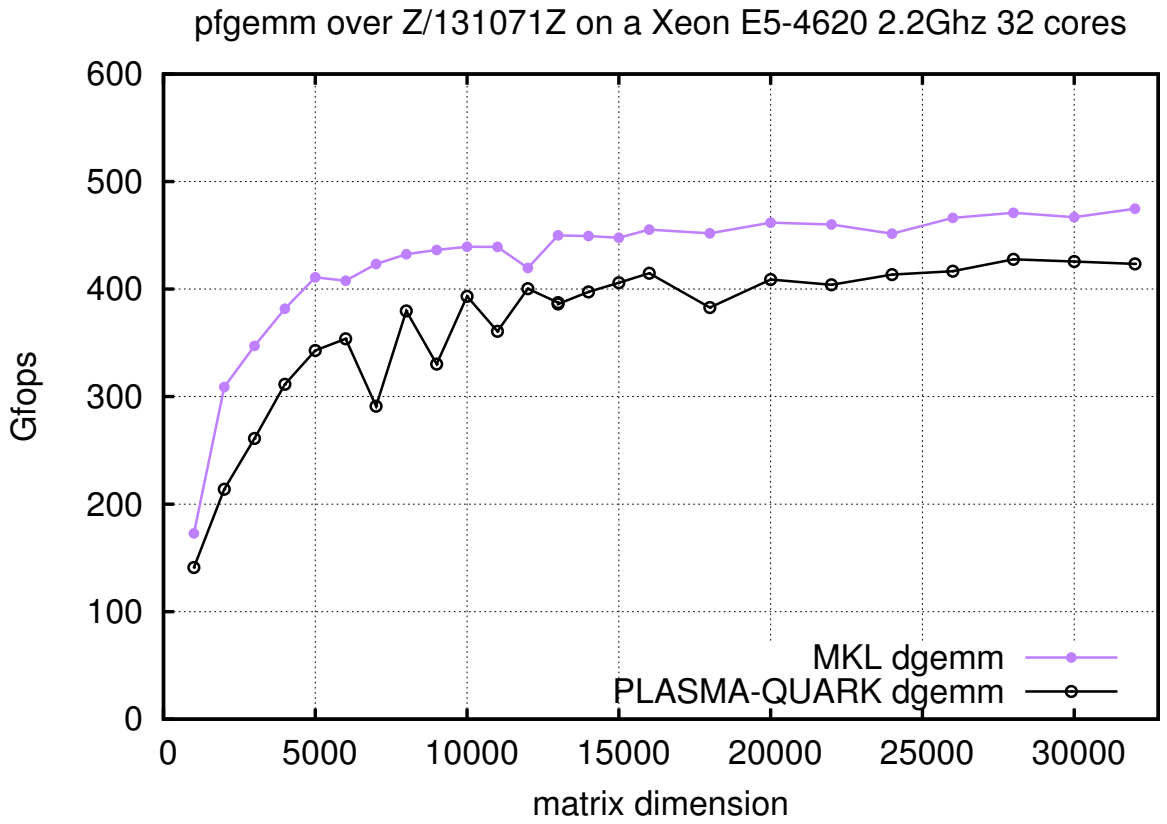
3 main models:

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

Parallel matrix multiplication

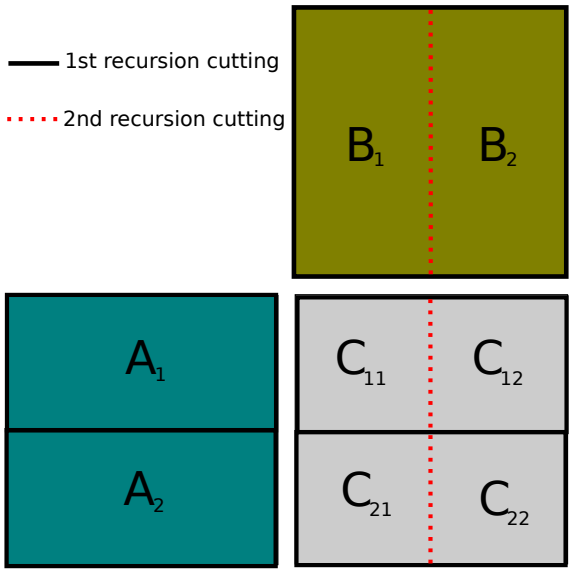
[Dumas, Gautier, P. & Sultan 14]

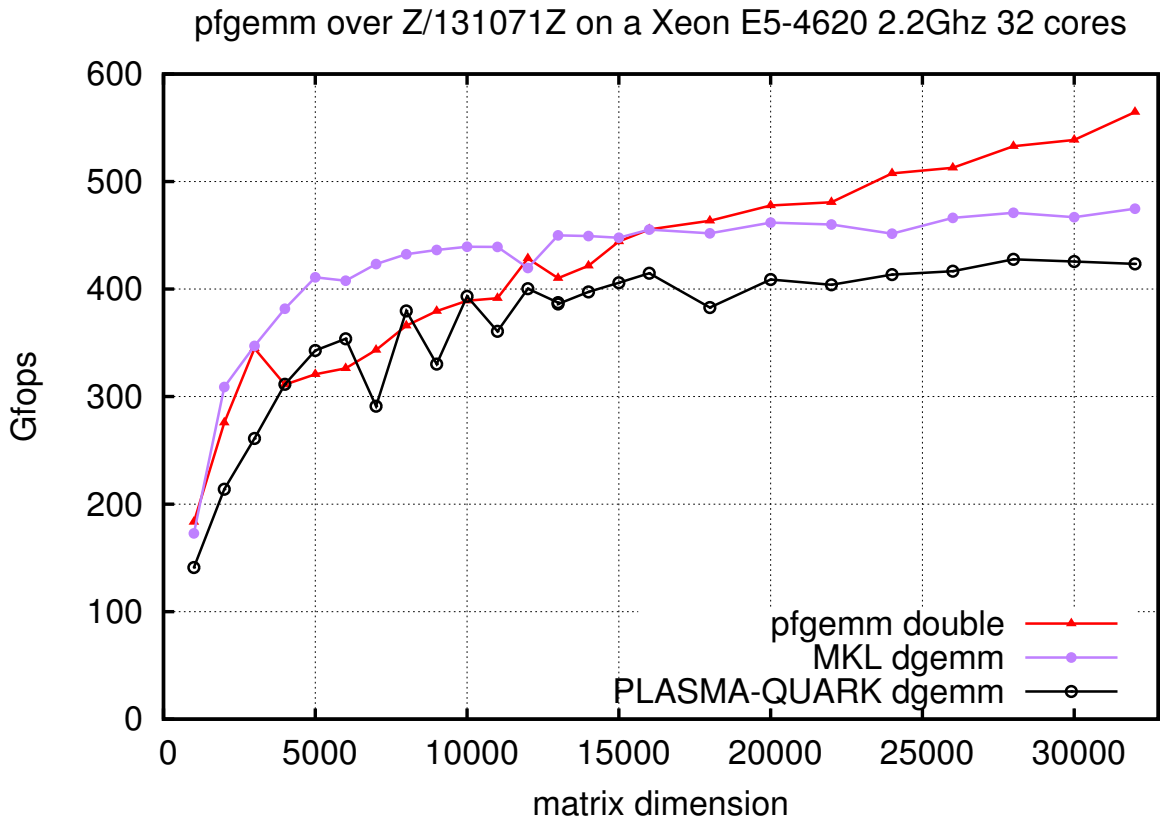




Parallel matrix multiplication

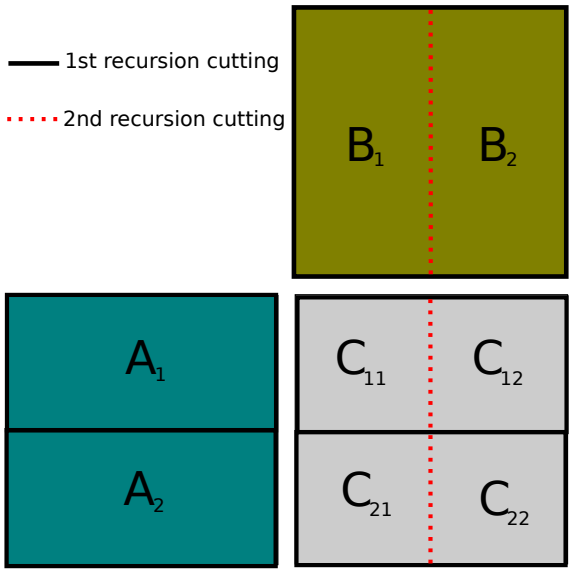
[Dumas, Gautier, P. & Sultan 14]

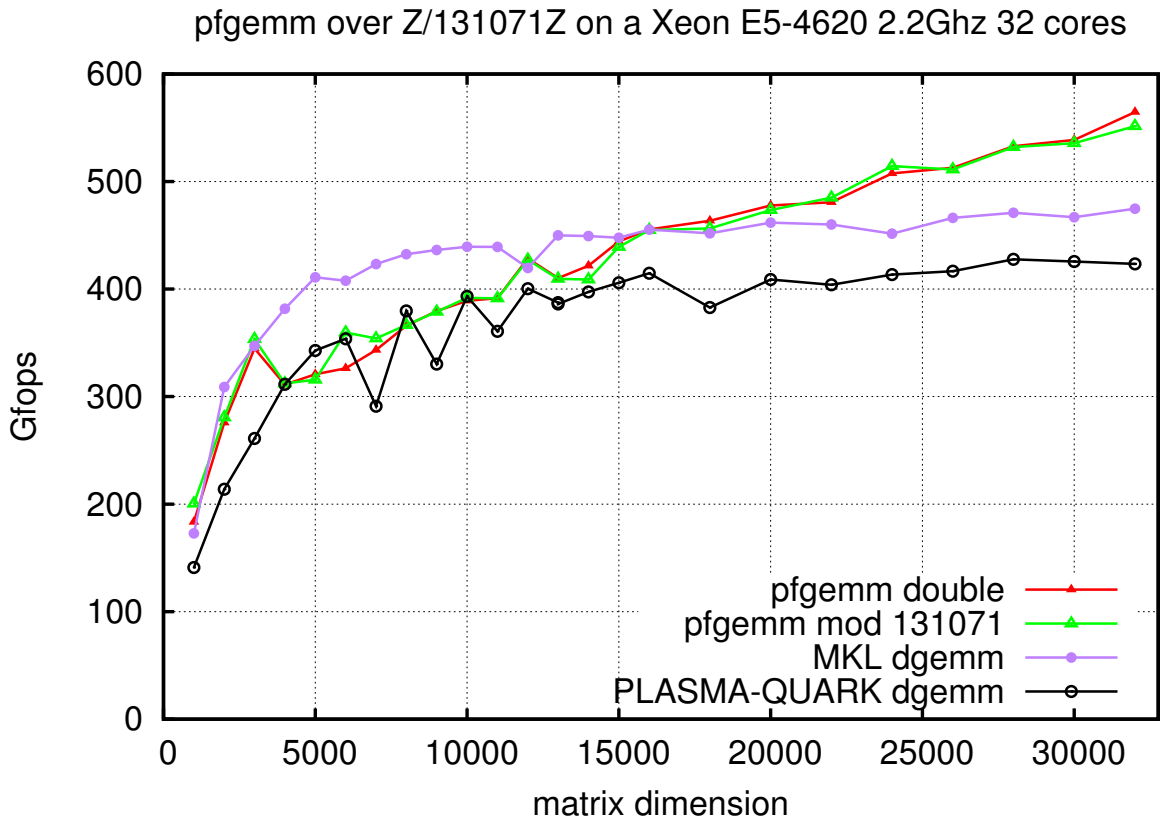




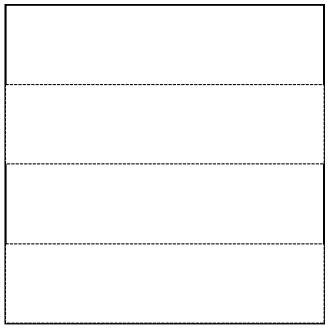
Parallel matrix multiplication

[Dumas, Gautier, P. & Sultan 14]



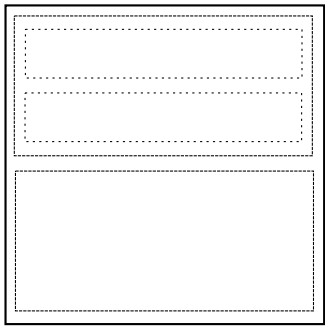


Gaussian elimination



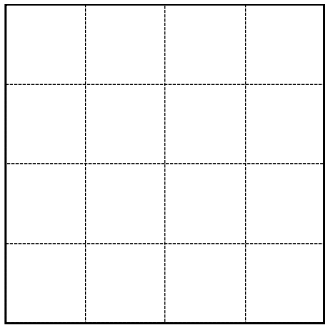
Slab iterative

`LAPACK`



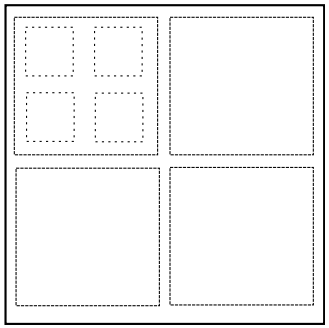
Slab recursive

`FFLAS-FFPACK`



Tile iterative

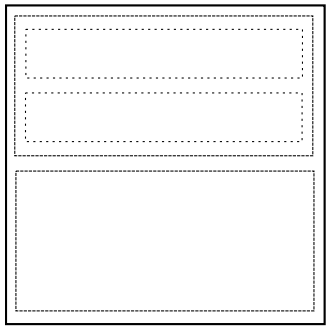
`PLASMA`



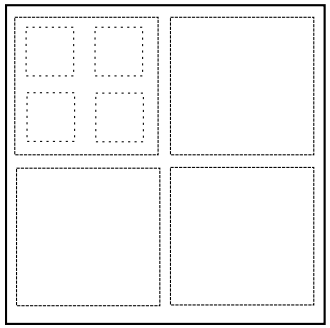
Tile recursive

`FFLAS-FFPACK`

Gaussian elimination



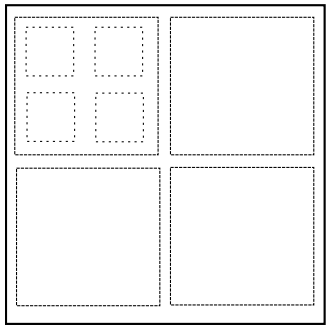
Slab recursive
FFLAS-FFPACK



Tile recursive
FFLAS-FFPACK

- Prefer recursive algorithms

Gaussian elimination



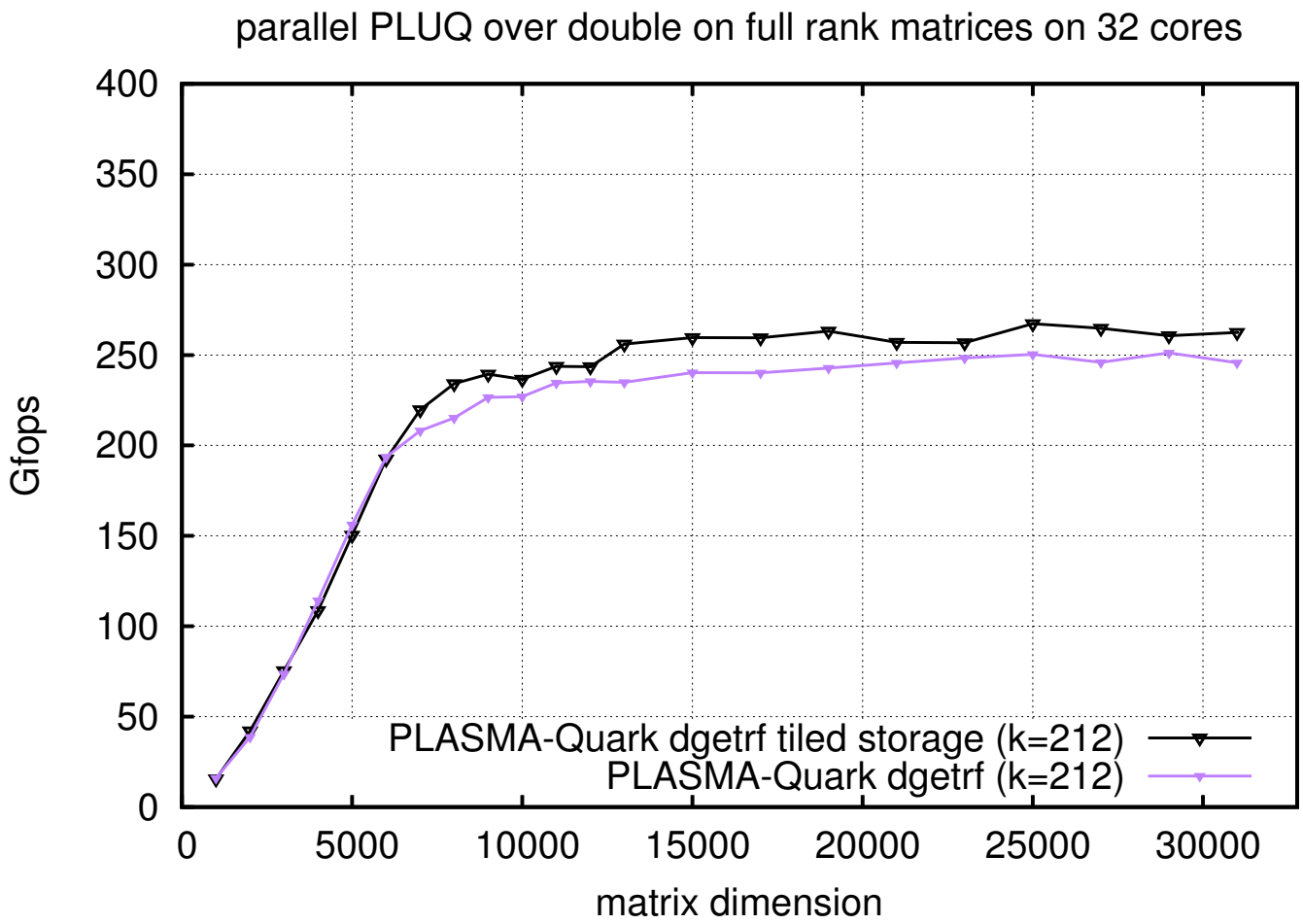
Tile recursive

`FFLAS-FFPACK`

- Prefer recursive algorithms
- Better data locality

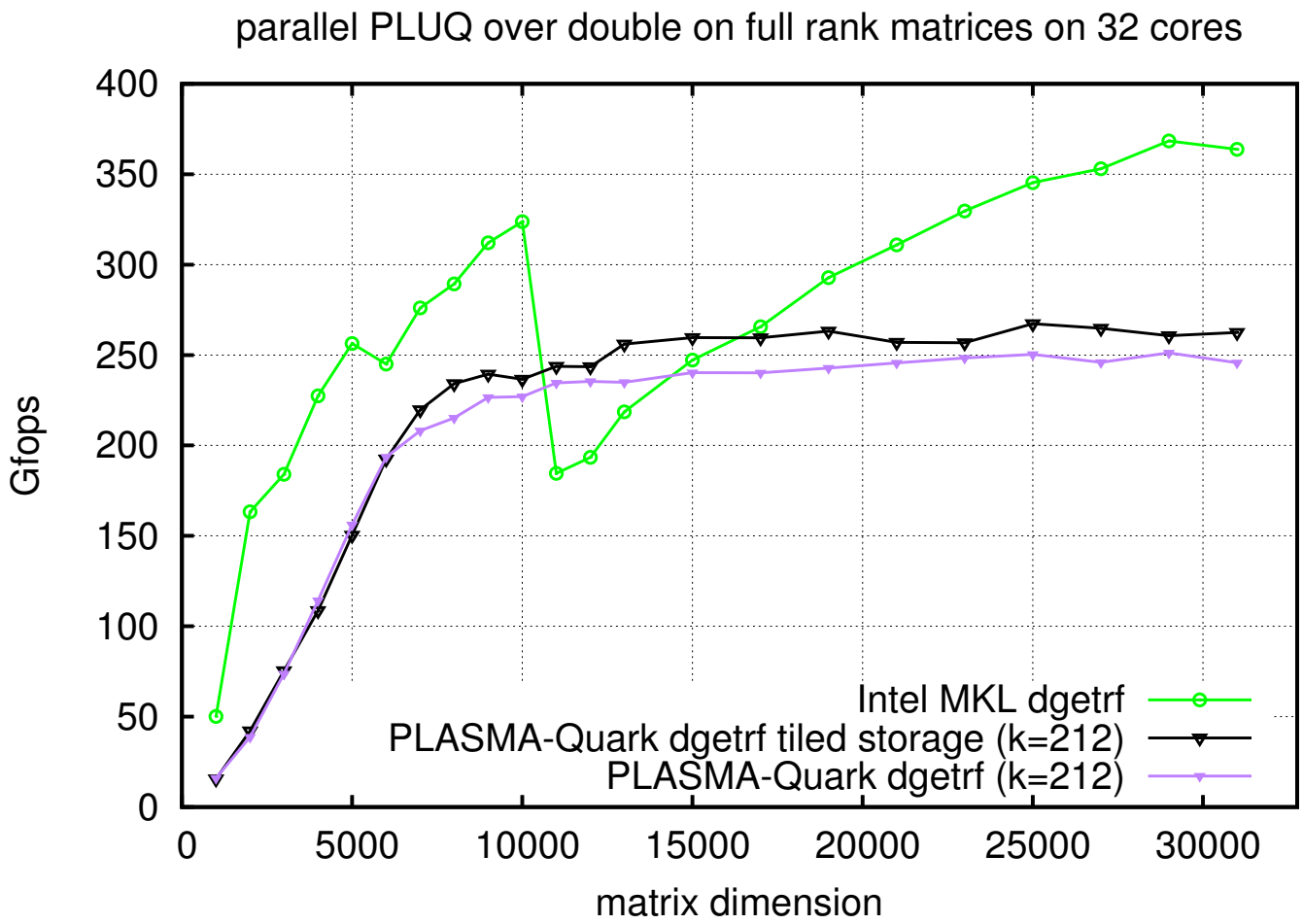
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over $\mathbb{R}$ (no modulo)



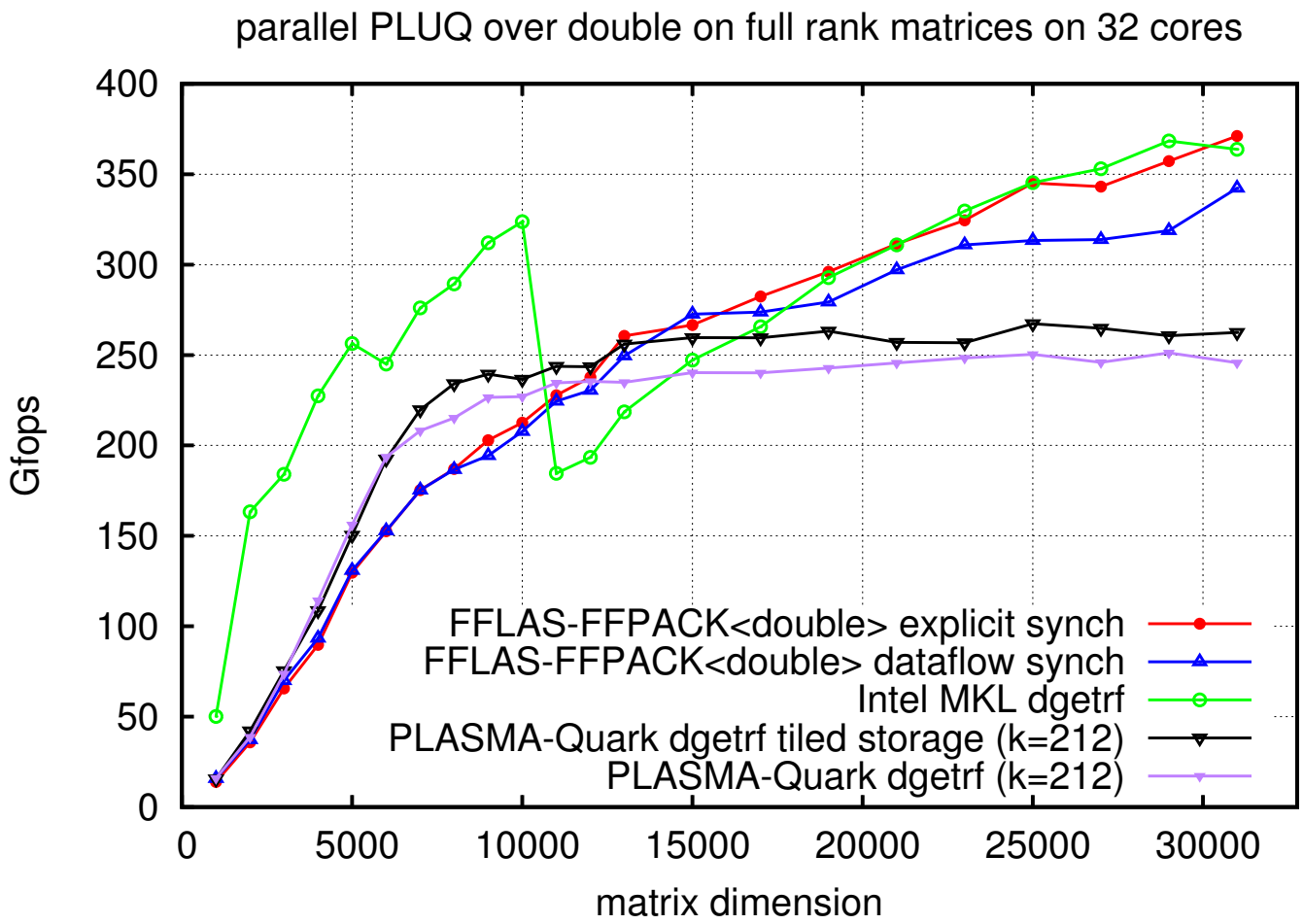
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over $\mathbb{R}$ (no modulo)



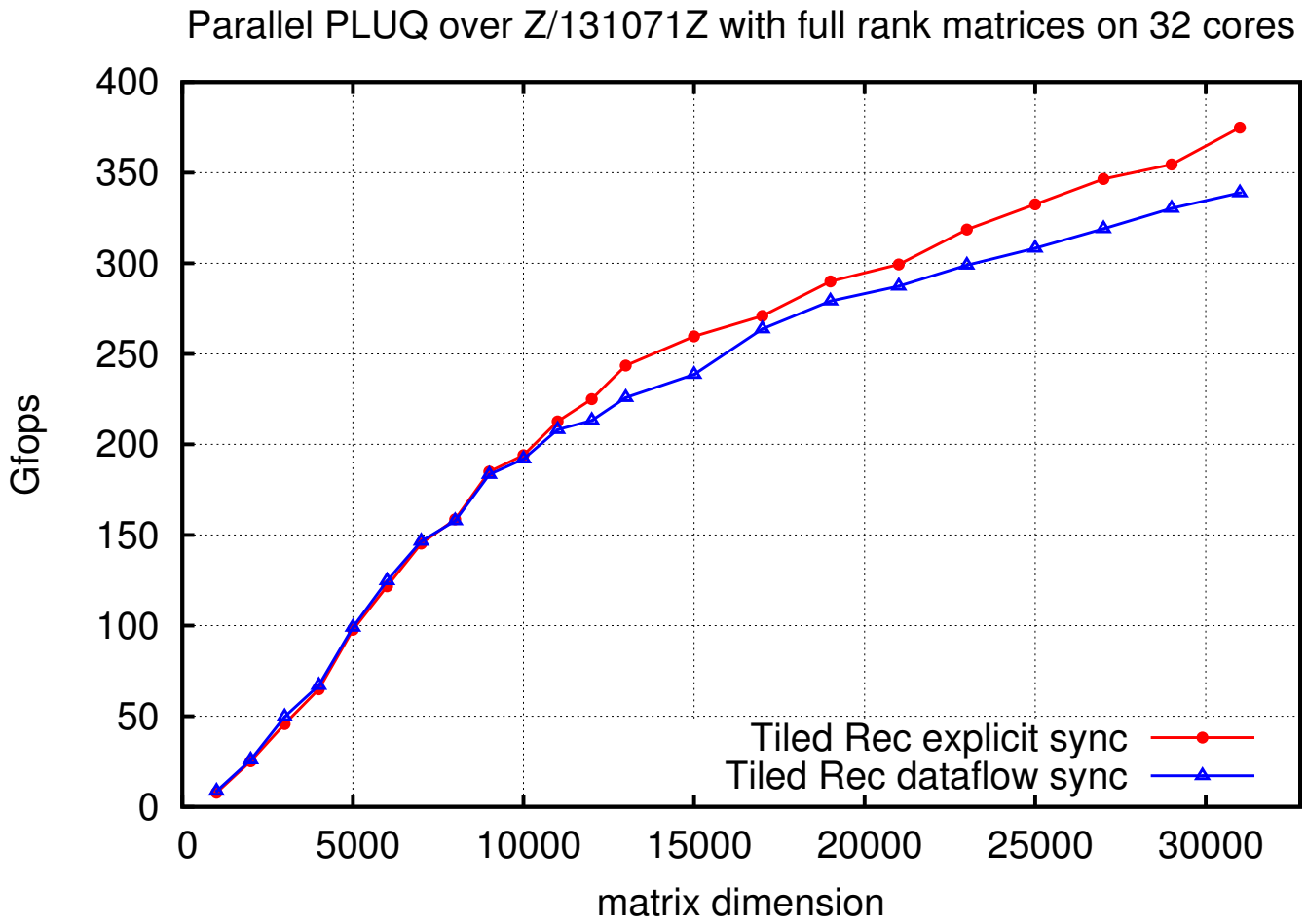
Full rank Gaussian elimination

[Dumas, Gautier, P. and Sultan 14] Over $\mathbb{R}$ (no modulo)



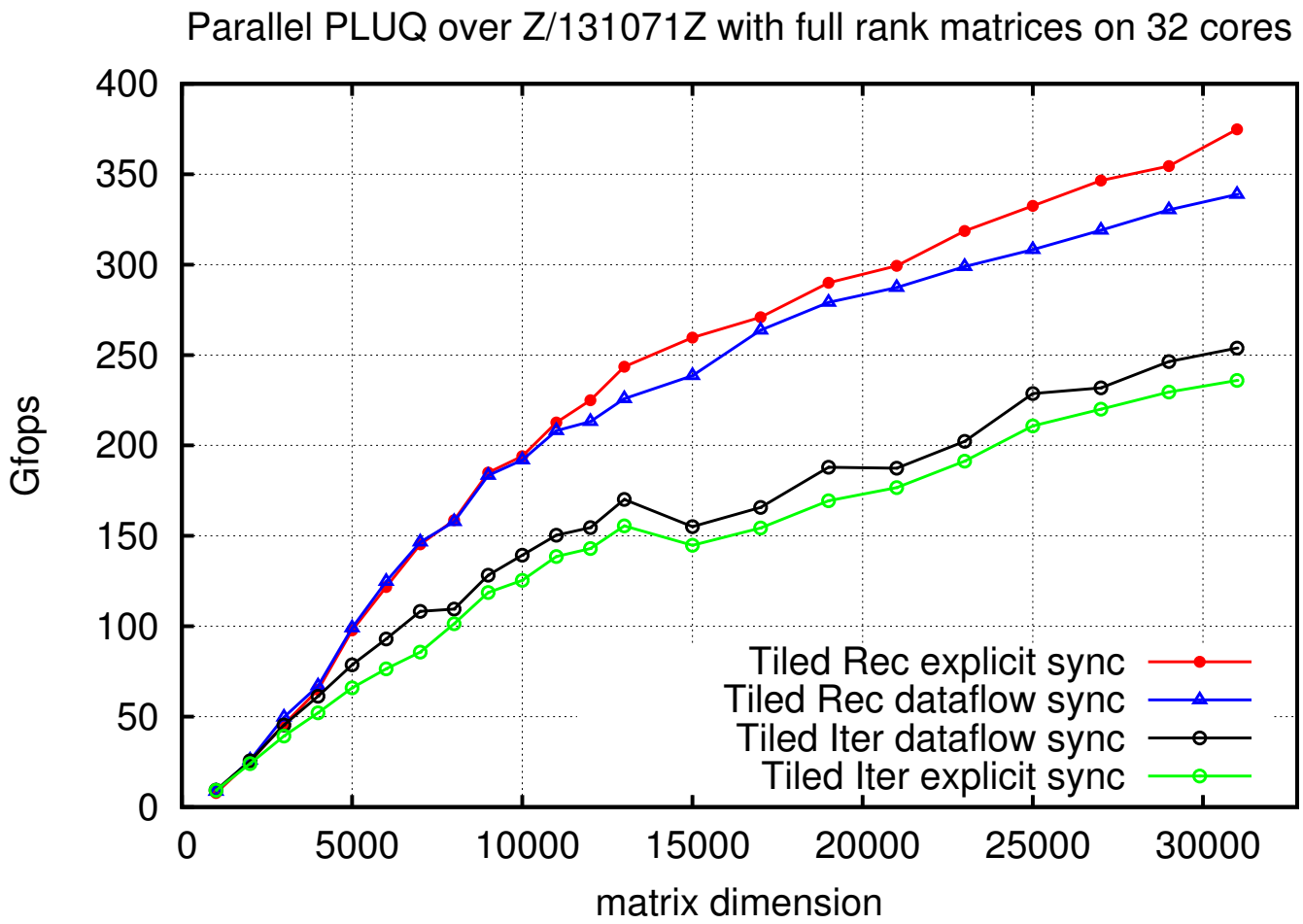
Full rank Gaussian elimination

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



Full rank Gaussian elimination

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



Perspectives

Large scale distributed exact linear algebra

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

Perspectives

Large scale distributed exact linear algebra

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

Structured linear algebra

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

Perspectives

Large scale distributed exact linear algebra

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

Structured linear algebra

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

Symbolic-numeric computation

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

Perspectives

Large scale distributed exact linear algebra

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

Structured linear algebra

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

Symbolic-numeric computation

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

Thank you