Simultaneous Modular Reduction and Kronecker Substitution for small finite fields

Jean-Guillaume Dumas
Laurent Fousse
Bruno Salvy

Grenoble University
Laboratoire Jean Kuntzmann
Applied Mathematics and Computer Science Department

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA
centre de recherche PARIS - ROCQUENCOURT

Small field dense linear algebra

- Integer Factorization, discrete logarithm
  - Linear algebra modulo 2, modulo $n=p_1.p_2 \ldots p_k$
- Combinatorics
  - Integer normal forms, integer minimal/characteristic polynomials
- Stable Algorithms for numerical problems
  - Rational matrices: Chinese reconstruction

- Sparse Matrices
  - Bloc methods (Coppersmith-Wiedemann, Lanczos) :
    - ⇒ dense blocks such that MM is fast
  - Probabilistic Methods (e.g. success depends on field size):
    - ⇒Resolution in a finite extension

[May, Saunders, Wan, ISSAC 2007]

- Study of difference sets and partial difference sets in algebraic design theory *[Weng,Qiu,Wang,Xiang 2007]*
- Requires computations of the rank of almost dense matrices of sizes 59049, 531441, 4782969, …
- Modulo small primes $p \equiv 3 \bmod 4$
  $p = 3, 7, 11, 19, 23, \ldots$

High performance / Exact computations?

- Memory : optimize memory accesses, cache usage etc.
  - cf numerical BLAS (ATLAS, GOTO, etc.)

- Exact computations (modular, finite fields, etc.)

  - Division (e.g.: modulo p) can be 10 to 100 times slower than machine multiplication/addition

  - SSE (128 bits registers): simultaneous arithmetic operations
  - in 2008, no integer multiplication available (only floating points)

FFLAS

[D, Gautier, Pernet 2002]

- Division management:
  - Homomorphism to Z: delay the modular reduction, compute a whole dot product before remaindering
- Locality management:
  - Blocks
- SSE usage:
  - Leave the linear algebra to a numerical code used exactly
- Integrated in Maple (LinearAlgebra:-Modular) and Magma since then

FFLAS: Exact linear algebra

Ex: Matrix multiplication mod a prime p

1. Convert matrices mod p towards floating point matrices (*double*)
2. Use numerical BLAS (e.g. GOTO) to multiply within floating point
3. Convert back the *doubles* modulo p

$O(n^2)$ conversions versus $O(n^3)$ fast arithmetic operations

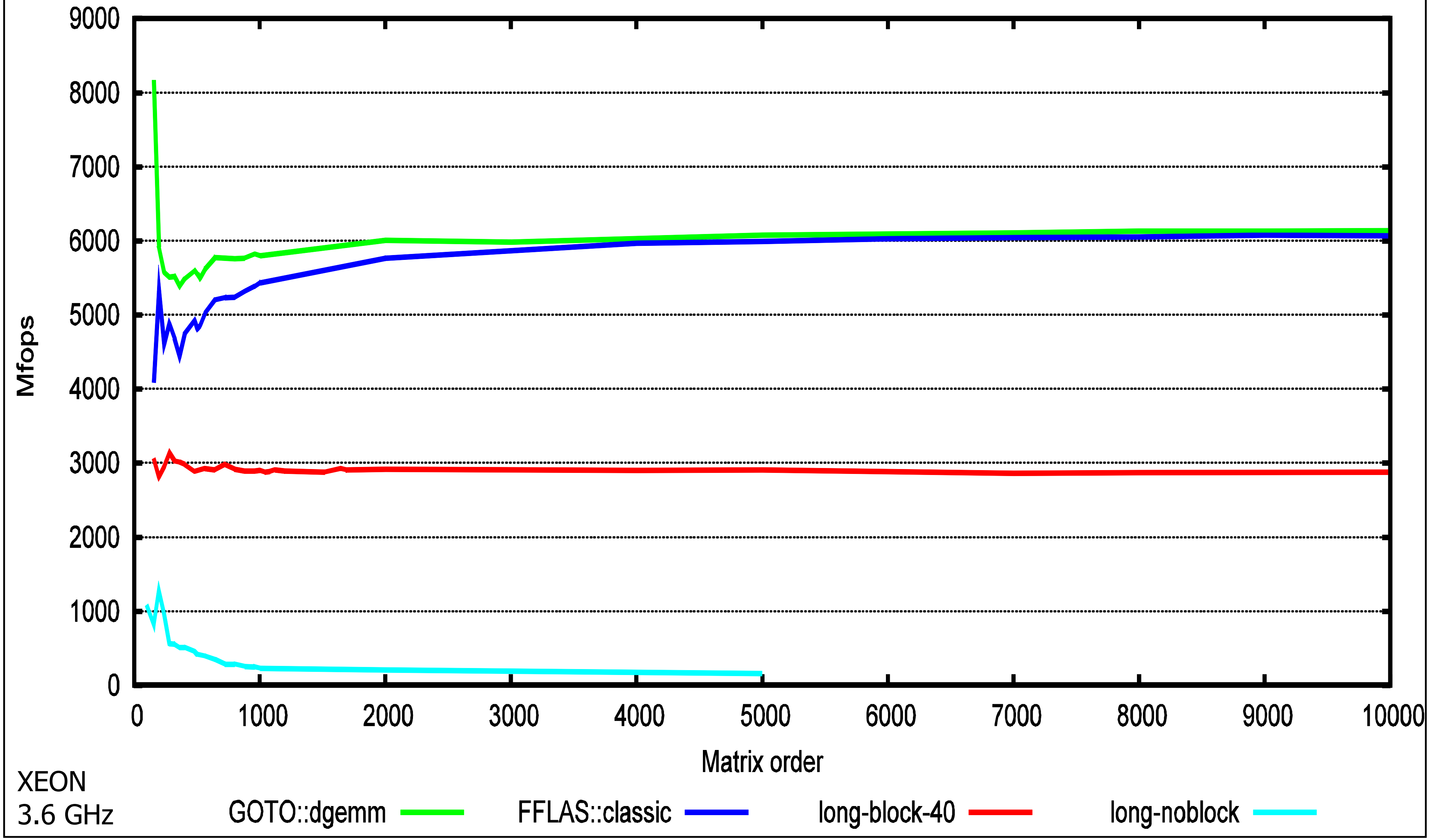
⇒ Exact as long as dot products do not overflow

⇒ Each one must fit inside the mantissa

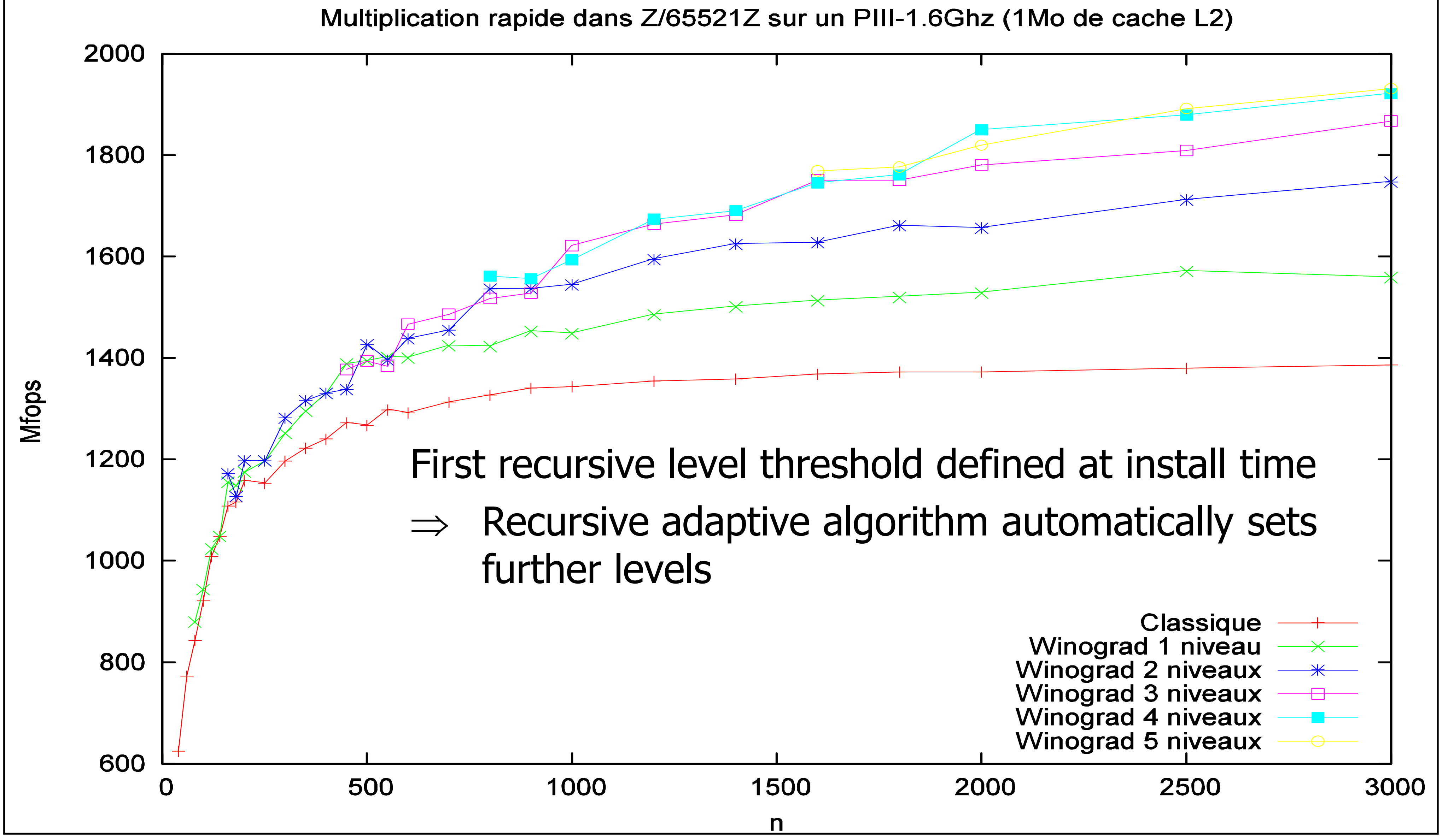
Ex.: $n\,(p-1)^2/4 < 2^{52}$: for $p \leq 2^{16}$, $n=4\,000\,000$ is OK
for $n \leq 6000$, modulo can have 20 bits

For larger primes or larger matrices, it is required to make the first recursive calls over the finite field and use the numerical routines only when the block is small enough.

Cache and SSE aware on a XEON 3.6 GHz



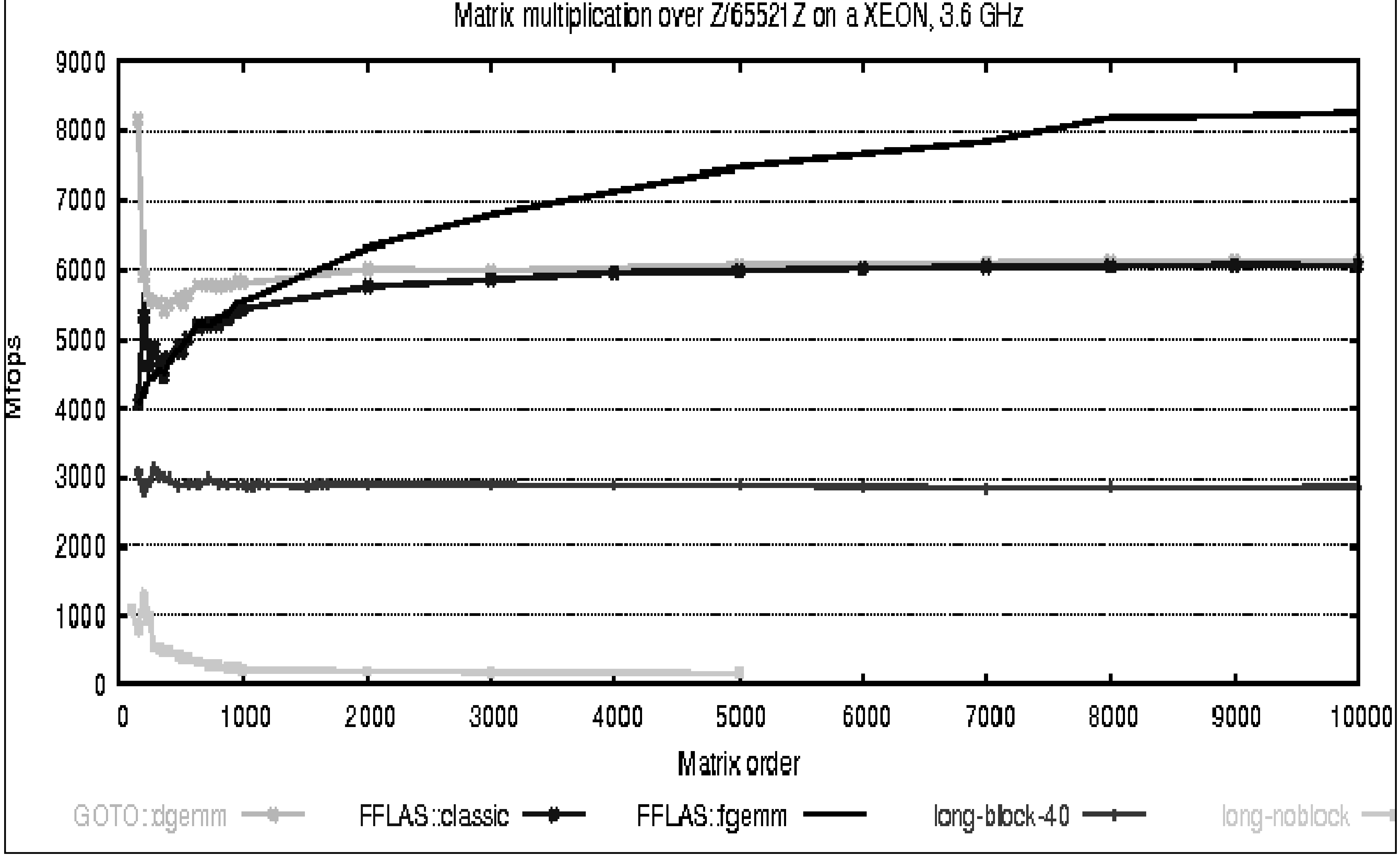
Strassen-Winograd Multiplication: 38%gain on a PIII 1.6 GHz



First recursive level threshold defined at install time

$\Rightarrow$ Recursive adaptive algorithm automatically sets further levels

FFLAS : 8.2 Gffops on a XEON 3.6 GHz



Compressed Arithmetic

0. Context

1. Compressed Arithmetic
    - Delayed reduction
    - Kronecker Substitution and polynomial multiplication
    - REDQ : Simultaneous Modular Reduction
    - Dot product

2. Modular Polynomial Multiplication

3. Modular Linear Algebra
    - Matrix Compression
    - REDQ with Left and Right Matrix Compression
    - Full Compression

4. Small Extension Field Linear Algebra

Delayed Modular Reduction

- Instead of computing a modulo p residue modulo p for each arithmetic operation:
  - Delayed the reduction after several +,* …

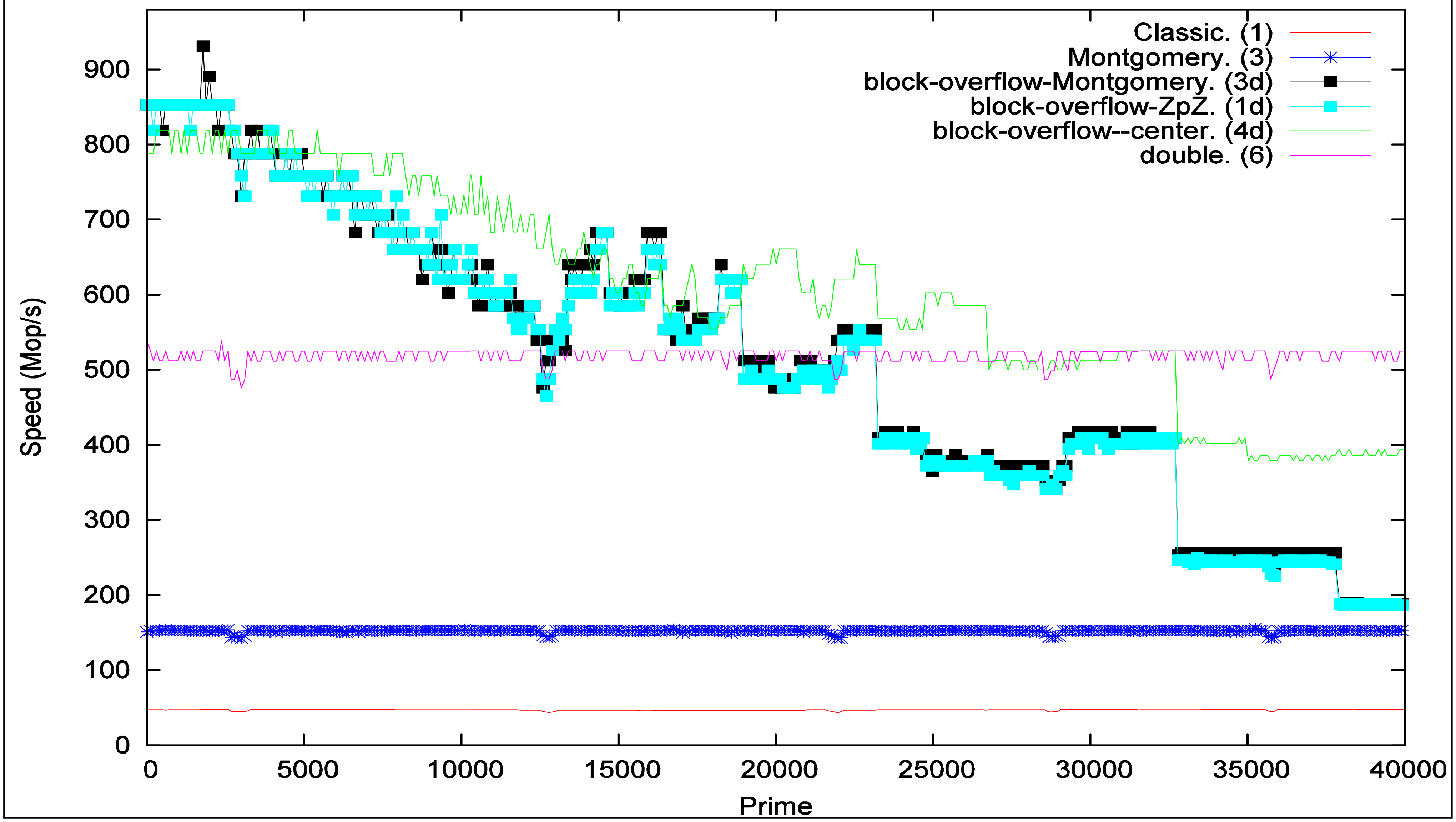
☺ Delayed reduction: if $k\ p^2 <$ wordsize then
  - At least k products are possible without overflow !
  - Block operations by k and reduce only once every k products

👍 Tricks *[D., Zimmermann 2004]*
  - Test every accumulation and reduce only in case of overflow
  - Make k operations first, and then only test for overflow
  - Replace division : $h = 2^{32} + x \Rightarrow h = x + CORR$,
    where $CORR = (2^{32}\ \%\ p)$
  - Use a centered modular representation
  - …

Delayed reduction Dotproduct

[D. 2004]

Dot product of a vector with 512 elements on a PIII 993 MHz



Compressed arithmetic?

- Within Z/2Z → binary implementations *NTL/M4RI*
- Within GF(5^3) → how to use only 7 bits per element ?
- In Z/5Z[X] → use just 3 bits per coefficient ?

- This talk: show that we can mimic binary/SSE behavior for small primes

👍 Use a Q-adic Transform (Kronecker substitution)
  - Change of representation → remplace the indeterminate by a sufficiently large integer q:
    - $X^4+2X^3+3X^2+4X+5$ modulo 7
    - $64^4+2.64^3+3.64^2+4.64+5 = 17314053$
    - $100^4+2.100^3+3.100^2+4.100+5=102030405$

Kronecker Substitution (Q-adic Transform)

- $Q=100$

- $A(X) = X + 1$ → $DQT(A) = 100 + 1 = 101$
- $B(X) = X + 2$ → $DQT(B) = 100 + 2 = 102$

- $DQT(A) . DQT(B) = 10302$

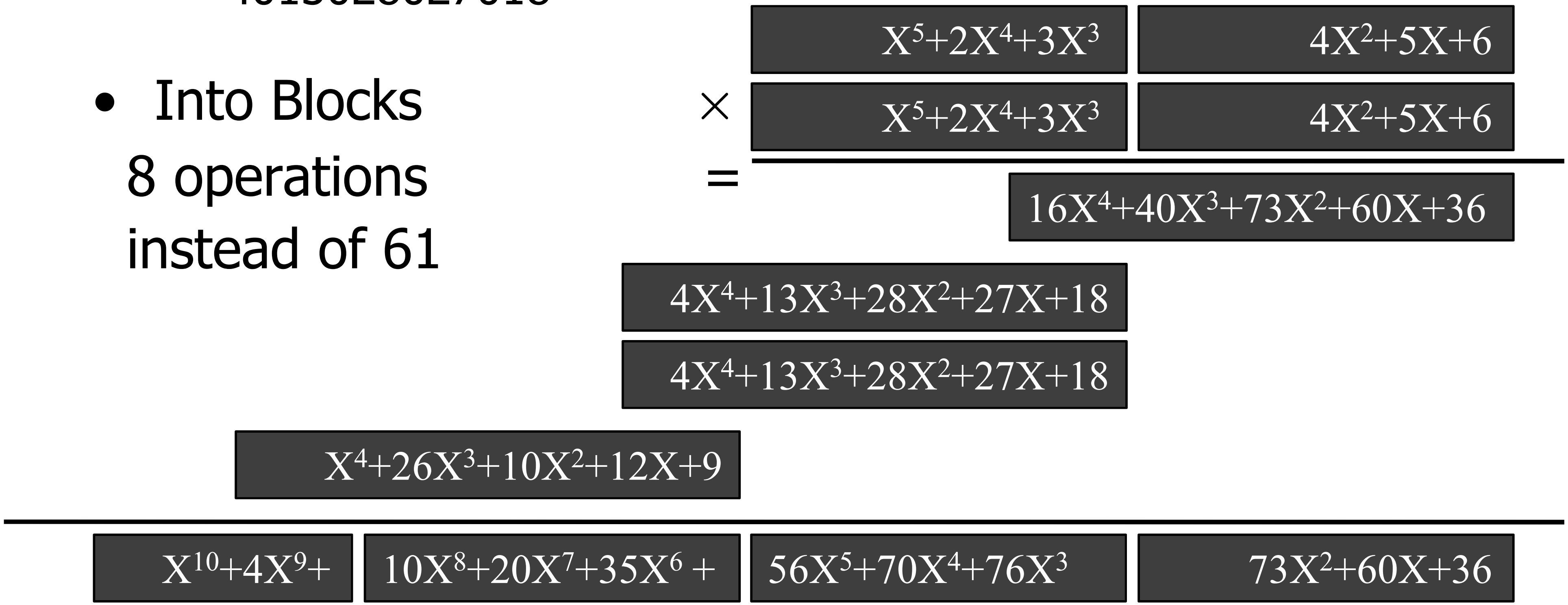
- $A \times B = X^2 + 3X + 2$

- $DQT(A \times B) = 100^2 + 3.100 + 2 = 10302$

Compressed polynomial multiplication

- Cut polynomials into blocks
  - E.g. [1,2,3] × [4,5,6], is replaced by 1002003 × 4005006 = 4013028027018
- Into Blocks

8 operations
instead of 61



Only problem: how to reduce, fast ?

1^{st} tool: Floating point division

- Euclidian division r= k.p +u
- How to compute k efficiently ?
  - Direct integer division is (very) expensive
  - [Shoup's NTL]: use floating point division
    - Precompute invp = 1.0/static_cast<double>(p);
    - Recover k by $\lfloor r * invp \rfloor$
- Problem: due to rounding approximations results could be off by one
- Algorithm: breaks pipeline with tests to correct the results

Improvements: play with rounding modes

- Change of rounding modes is costly, still
  - 1 rounding mode for precomputations
  - 1 rounding mode for algorithms
- Three rounding modes:
  - ▲ (upward) ; ▼ (downward) ; ♦ (nearest)
- Benefits and drawbacks
  - rounding 1/p upward ensures that result is off only upward → only 1 test instead of two
  - Validity range is modified

Quotients with different rounding modes

inv	mul	Range	Bound on r	Lost bits
$\triangle(\cdot)$	$\triangle(\cdot)$	$k \le \lfloor x \rfloor \le k+1$	$2^\beta/(4+2^{2-\beta})$	3
$\triangle(\cdot)$	$\diamond(\cdot)$	$k \le \lfloor x \rfloor \le k+1$	$2^\beta/(3+2^{1-\beta})$	2
$\triangle(\cdot)$	$\triangledown(\cdot)$	$k \le \lfloor x \rfloor \le k+1$	$2^\beta/2$	1
$\diamond(\cdot)$	$\triangle(\cdot)$	$k \le \lfloor x \rfloor \le k+1$	$2^\beta/(3+2^{1-\beta})$	2
$\diamond(\cdot)$	$\diamond(\cdot)$	$k-1 \le \lfloor x \rfloor \le k+1$	$2^\beta/(2+2^{-\beta})$	2
$\diamond(\cdot)$	$\triangledown(\cdot)$	$k-1 \le \lfloor x \rfloor \le k$	–	0
$\triangledown(\cdot)$	$\triangle(\cdot)$	$k-1 \le \lfloor x \rfloor \le k+1$	$2^\beta/2$	1
$\triangledown(\cdot)$	$\diamond(\cdot)$	$k-1 \le \lfloor x \rfloor \le k$	–	0
$\triangledown(\cdot)$	$\triangledown(\cdot)$	$k-1 \le \lfloor x \rfloor \le k$	–	0

2nd tool: Montgomery reduction REDC

- System division replaced by shifts and masks

```
#define MASK 65535UL
#define B 65536UL
#define HALF_BITS 16
/* nim is precomputed to -1/p mod B
with the extended gcd */
```

AXPY:

```
1.  c = (a × x + y);
```

REDC:

```
2.  unsigned long c0 (c & MASK);     /*  c     mod B           */
3.  c0 = (c0 * nim) & MASK;          /*  -c/p  mod B           */
4.  c += c0 * p;                     /*  c = 0 mod B
                                         c = ax+y mod p        */
5.  c >>= HALF_BITS;                 /*  high bits of c        */
```

CORRECTION:

```
6.  return (c>p?c-p:c);              /*  0 < c < 2p            */
```

REDQ: simultaneous reduction *[D. 2008]*

- <u>How to compute k modular reductions simultaneously ?</u>

👍 Floating point reduction [Shoup]:
  - $r = r - \lfloor r/p \rfloor \times p$

👍 [Montgomery] reduction (REDC):
  - Use divisions by powers of 2 to avoid division by p

$\Rightarrow$ Combine floats/REDC on the DQT:

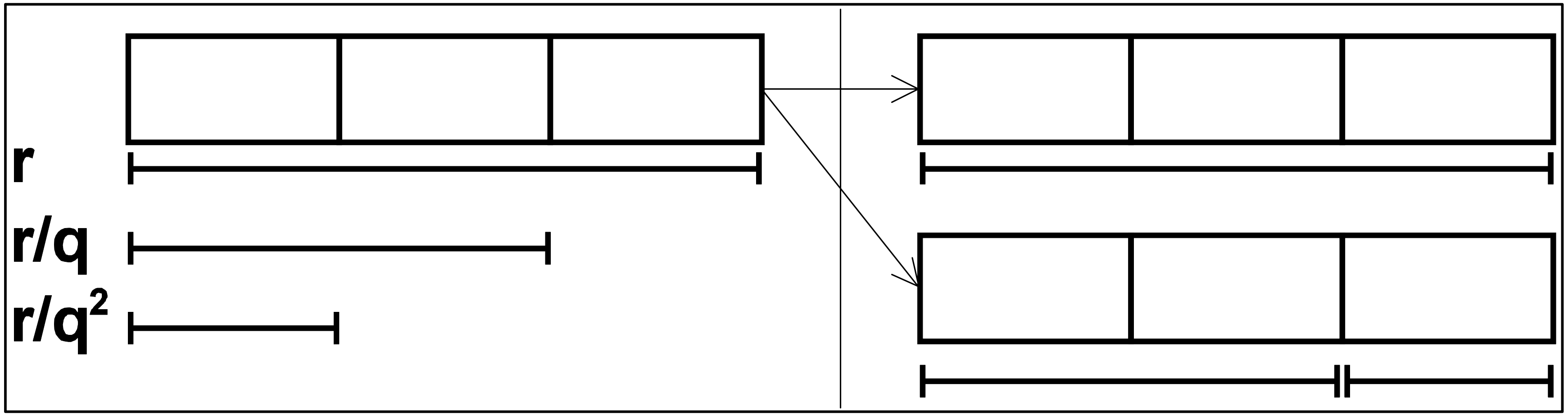
REDQ_COMPRESSION

1. Compute a SINGLE division : $d = \lfloor r/p \rfloor$
2. "Shift"/multiply k times : $u_i = r/q^i - \lfloor d/q^i \rfloor \times p$

REDQ_CORRECTION when required

3. Adjust : $r_i = (u_i - qu_{i+1}) \bmod p$

Binary case: packing multiplications



- After each iterations **log$_2$(q)** bits needs to be discarded
  - ⇒Recopy parts of r into several words
  - ☺ Only $\lceil k/2 \rceil$ axpy required

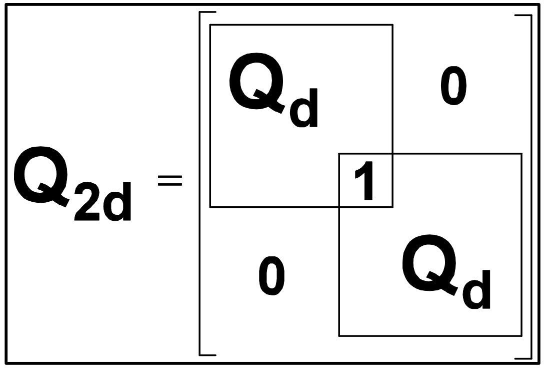
Fast REDQ: tabulate CORRECTION

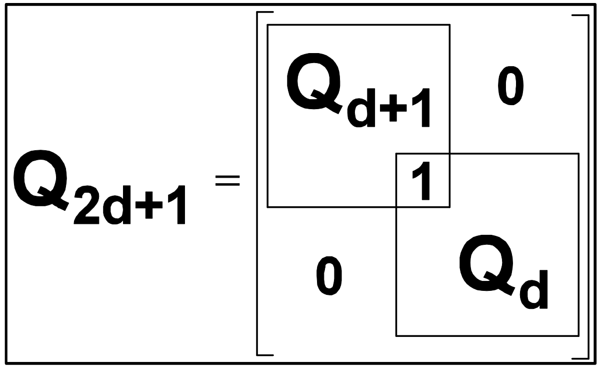
- CORRECTION is slow (back to k divisions !)
- But all the u_i are smaller than p thanks to the COMPRESSION

$\Rightarrow$ Adjustment is tabulated: $r = Q_d\, u \bmod p$

$$Q_d = \begin{bmatrix} 1 & -q & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -q \\ 0 & \ldots & \ldots & 0 & 1 \end{bmatrix} = \left[\begin{array}{c} \underline{Q}_d \\ \hline 0 \ldots 0\ 1 \end{array}\right]$$

Fast REDQ: time-memory trade-off





Memory	Time
0	d (mul,add,mod)
p^2	d accesses
p^i	$\left\lceil \frac{d}{i-1} \right\rceil$ accesses
p^{d+1}	1 access

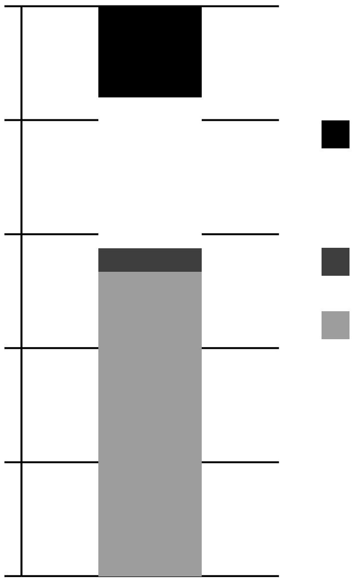
TMTO example:

```
Algorithm 2 $Q_6$ with an extra memory of size $p^3$
Input: $[u_0 \ldots, u_6] \in (\mathbb{Z}/p\mathbb{Z})^7$;
Input: a table $\underline{Q}_2$ of the associated $2 \times 3$ matrix-vector
    multiplication over $\mathbb{Z}/p\mathbb{Z}$.
Output: $[\mu_0, \ldots, \mu_6]^T = Q_6[u_0 \ldots, u_6]^T$.
  $a_0, a_1 = \underline{Q}_2[u_0, u_1, u_2]^T$;
  $b_0, b_1 = \underline{Q}_2[u_2, u_3, u_4]^T$;
  $c_0, c_1 = \underline{Q}_2[u_4, u_5, u_6]^T$;
  Return $[\mu_0, \ldots, \mu_6]^T = [a_0, a_1, b_0, b_1, c_0, c_1, u_6]^T$;
```

REDQ implementation efficiency

- Simultaneous reductions timings:

  Profiling $REDQ_5$:
  - ⓘ faster than five divisions …
  - ⓘ But 58% of the time was spent in type casts



- Solution: include some casts in the REDQ_CORR table
  - ☺ 54% gain
  - ☹ size of k-REDQ_CORR multiplied by k

32 bits fast 3-REDQ

```
inline void REDQ_COMP(UINT32_three& res, const double r, const double p){
    _ULL64_unions r_ll_copy, t_ll_copy;                  // union of 64, 17-34 or 34-17 bits

    r_ll_copy._64 = static_cast<UINT64>( r );
    t_ll_copy._64 = static_cast<UINT64>( r/p );                          // One float division
    res.high = static_cast<UINT32>(r_ll_copy._64-t_ll_copy._64*p); // One axpy
    r_ll_copy._17_34.low = r_ll_copy._34_17.high;                       // Packing
    t_ll_copy._17_34.low = t_ll_copy._34_17.high;                       // Packing
    r_ll_copy._64 -= t_ll_copy._64*p;                                   // Two axpy in one
    res.mid = static_cast<UINT32>(rll._17_34.high);
    res.low = r_ll_copy._17_34.low;
}

inline void REDQ3_CORR(UINT32_three& res, const Container<_UINT32 >& Q3)
{
    res._32=Q3[res._32];
}
```

Compressed Arithmetic

0. Context

1. Compressed Arithmetic
   - Delayed reduction
   - Kronecker Substitution and polynomial multiplication
   - REDQ : Simultaneous Modular Reduction
   - Dot product

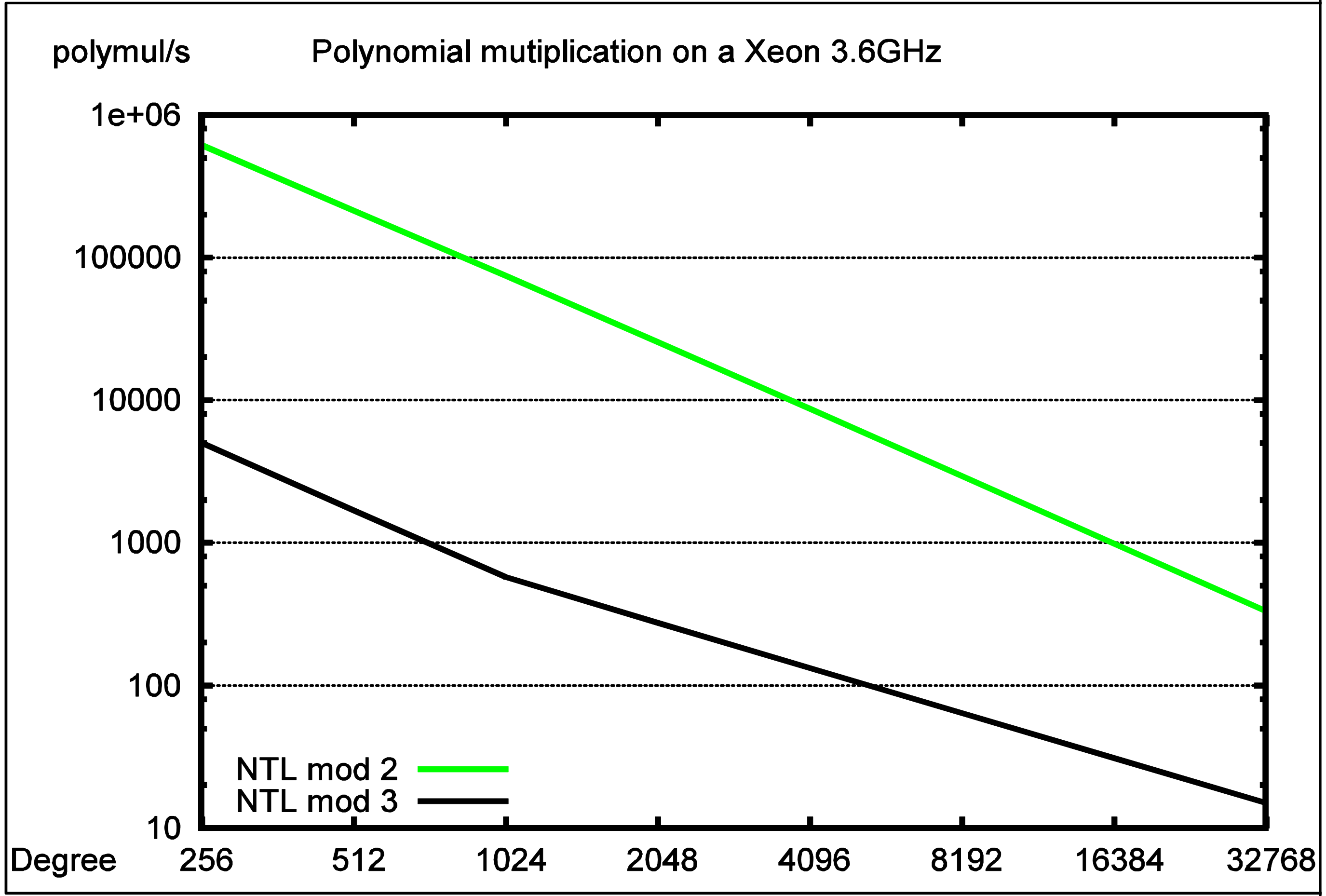
2. Modular Polynomial Multiplication

3. Modular Linear Algebra
   - Matrix Compression
   - REDQ with Left and Right Matrix Compression
   - Full Compression

4. Small Extension Field Linear Algebra

NTL polynomial multiplication

- Characteristic 2 : use of bits as elements
- Odd characteristic ?

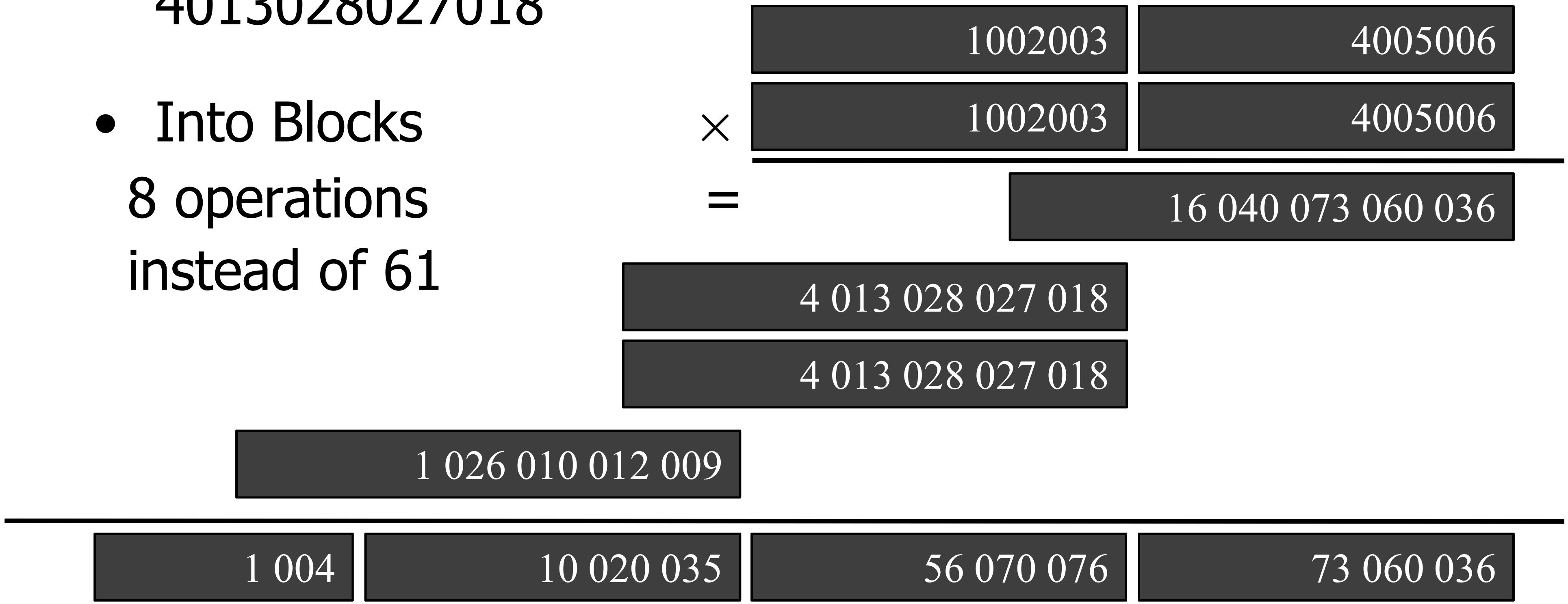


Compressed polynomial multiplication

- Cut polynomials into blocks
- [1,2,3] × [4,5,6], replaced by 1002003 × 4005006 = 4013028027018

- Into Blocks

8 operations
instead of 61



Then Reduce each bloc using REDQ

Complexity

- P of degree N in X → P of degree D_q in $Y=X^{d+1}$

$$D_q = \left\lceil \frac{N+1}{d+1} \right\rceil - 1$$

$$n_d = \left\lfloor \frac{2^{\beta+1}}{(p-1)^2} \right\rfloor \quad ; \quad n_q = \left\lfloor \frac{q}{(d+1)(p-1)^2} \right\rfloor$$

	Mul & Add	Reductions
Delayed	$(2N+1)^2$	$(2N+1)\left\lceil \frac{2N+1}{n_d} \right\rceil$ REDC
d-FQT	$(2D_q+1)^2$	$(2D_q+1)\left\lceil \frac{2D_q+1}{n_q} \right.$ REDQ_{2d+1}

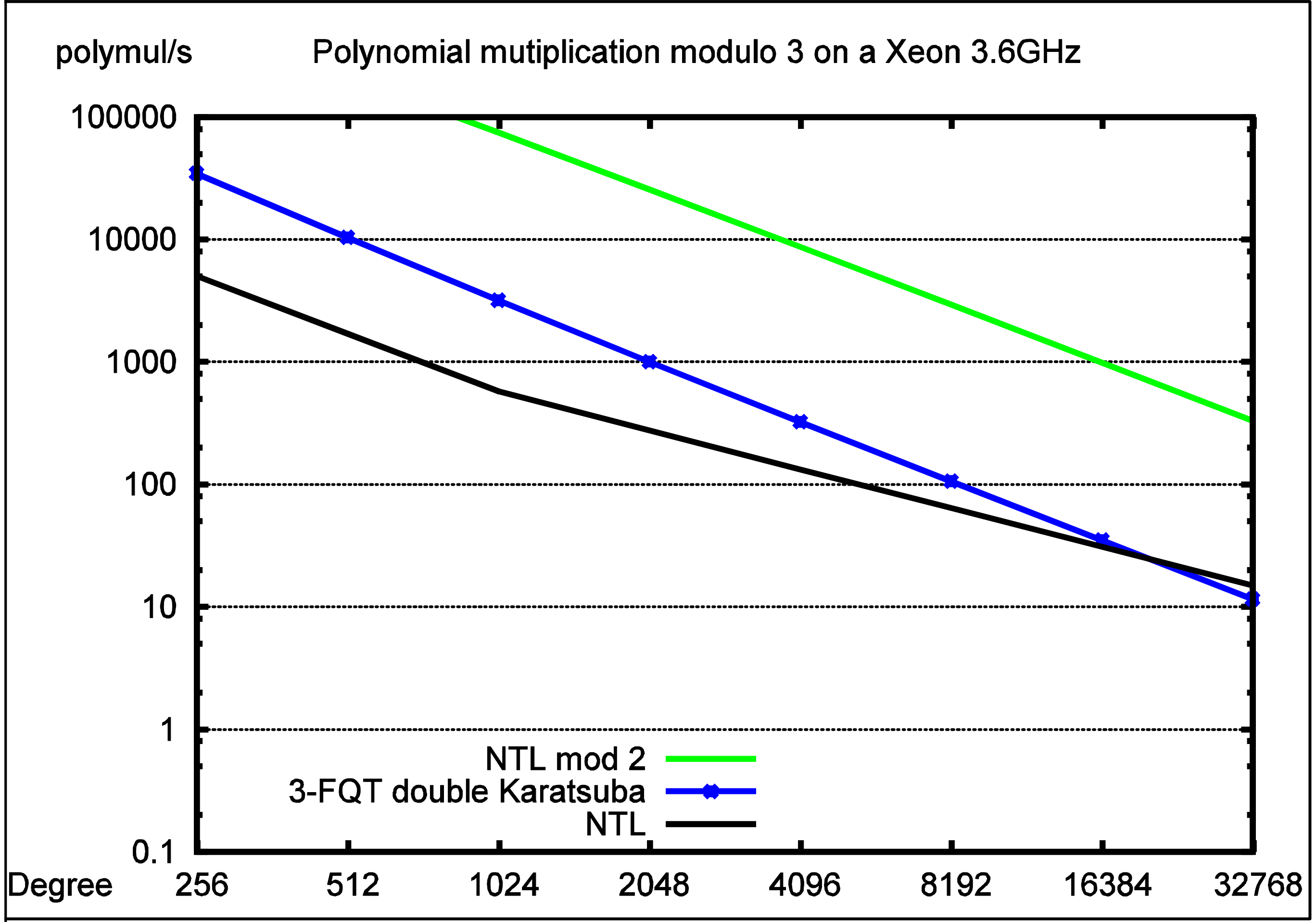
Example

- Degree N=500
- prime p=3
- Kronecker substitution with 4 elements per block
  - $D_q = 125$
  - $n_q = 11$
  - $n_d = 4.5\ 10^{16} >> N$

Algorithm	Mul & Add	Reductions
Delayed	10^6	10^3
4-FQT (floats, tabulations)	$8.6 \cdot 10^4$	$5.7 \cdot 10^3$

Modular polynomial multiplication

- Compressed arithmetics + Delayed reduction



- Classical algorithm
- Karatsuba with recursive threshold
- NTL is using FFT

Compressed Arithmetic

0. Context

1. Compressed Arithmetic
    - Delayed reduction
    - Kronecker Substitution and polynomial multiplication
    - REDQ : Simultaneous Modular Reduction
    - Dot product

2. Modular Polynomial Multiplication

3. Modular Linear Algebra
    - Matrix Compression
    - REDQ with Left and Right Matrix Compression
    - Full Compression

4. Small Extension Field Linear Algebra

Linear algebra with Q-adic Transform

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

$$\begin{bmatrix} Qa + b \\ Qc + d \end{bmatrix} \times \begin{bmatrix} e + Qg & f + Qh \end{bmatrix} =$$

$$\begin{bmatrix} * + (ae + bg)Q + * Q^2 & * + (af + bh)Q + * Q^2 \\ * + (ce + dg)Q + * Q^2 & * + (cf + dh)Q + * Q^2 \end{bmatrix}$$

Lower bound on Q

- Each multiplication is $\leq (p-1)^2$
- Polynomial of degree d
  - d+1 coefficient per machine word
  - Compression factor of (d+1)
  - Each polynomial coefficient is $\leq (d+1)(p-1)^2$
- Q-adic transform gets correct values by polynomial multiplication if

$$(d+1)(p-1)^2 < Q$$

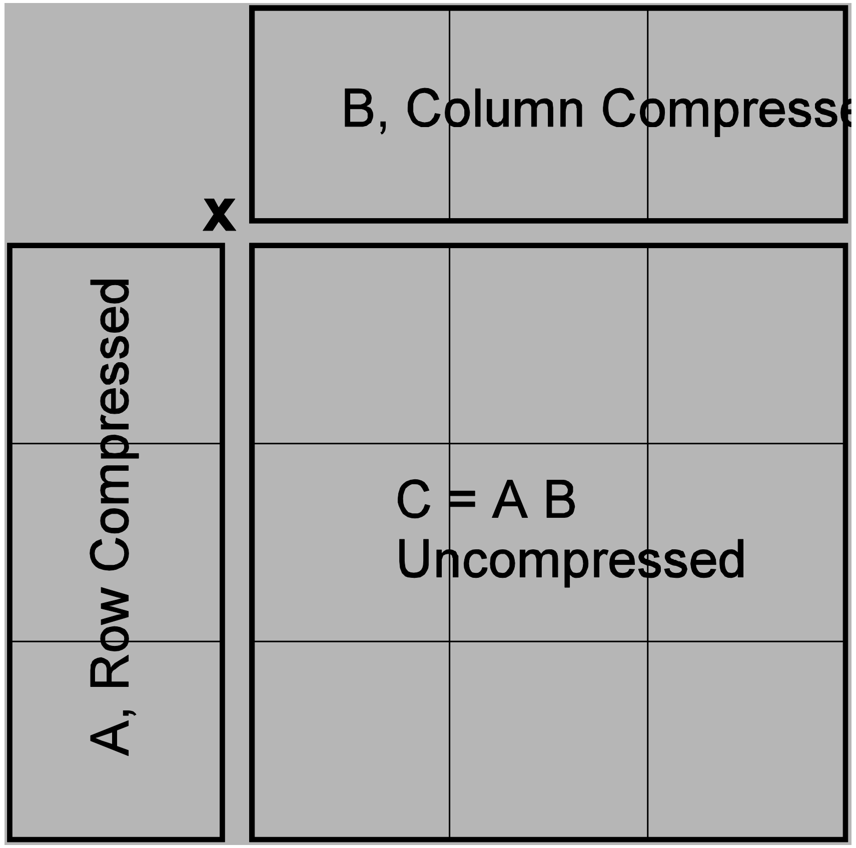
Can also use Delayed reduction

- Compression factor of d+1
- For a row of size k, use k/(d+1) machine words (k/(d+1) polynomials of degree d)
- Result is correct if intermediate coefficients do not overflow Q:

$$\frac{k}{d+1}(d+1)(p-1)^2 = k(p-1)^2 < Q$$

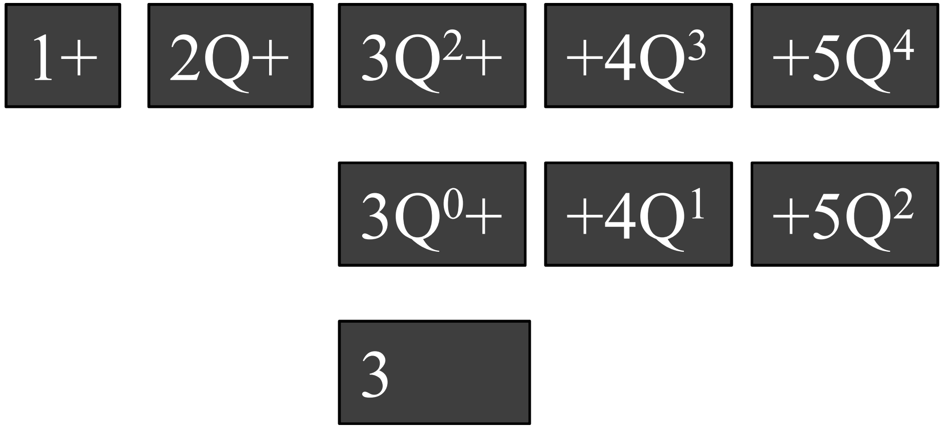
Algorithm CMM

1. CA = CompressReverseRows(A);
2. CB = CompressColumns(B);
3. C = CA × CB
4. Coefficient recovery:
    1. Get the middle degree term
    2. Compute one remainder



Middle degree term recovery

- Q-adic polynomial stored in a machine word
- Shift
- Mask



$1+$ $2Q+$ $3Q^2+$ $+4Q^3$ $+5Q^4$

$3Q^0+$ $+4Q^1$ $+5Q^2$

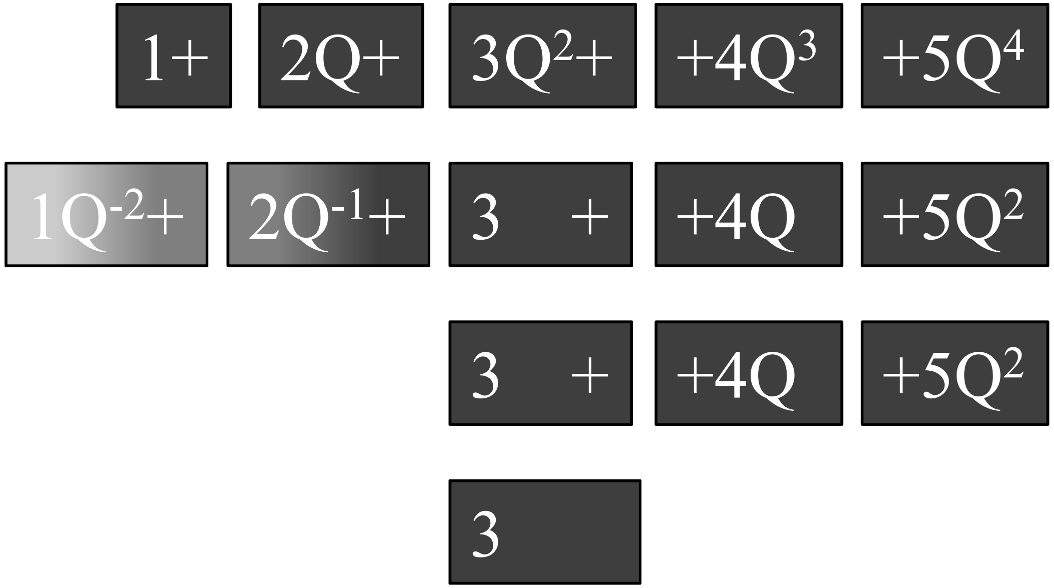
3

- Lower bits (1+2Q) are not required

⇒ floating point precision

Available mantissa and upper bound on Q

- Q-adic polynomial stored in a machine word
- Floating point precision
- Shift/Floor
- Mask

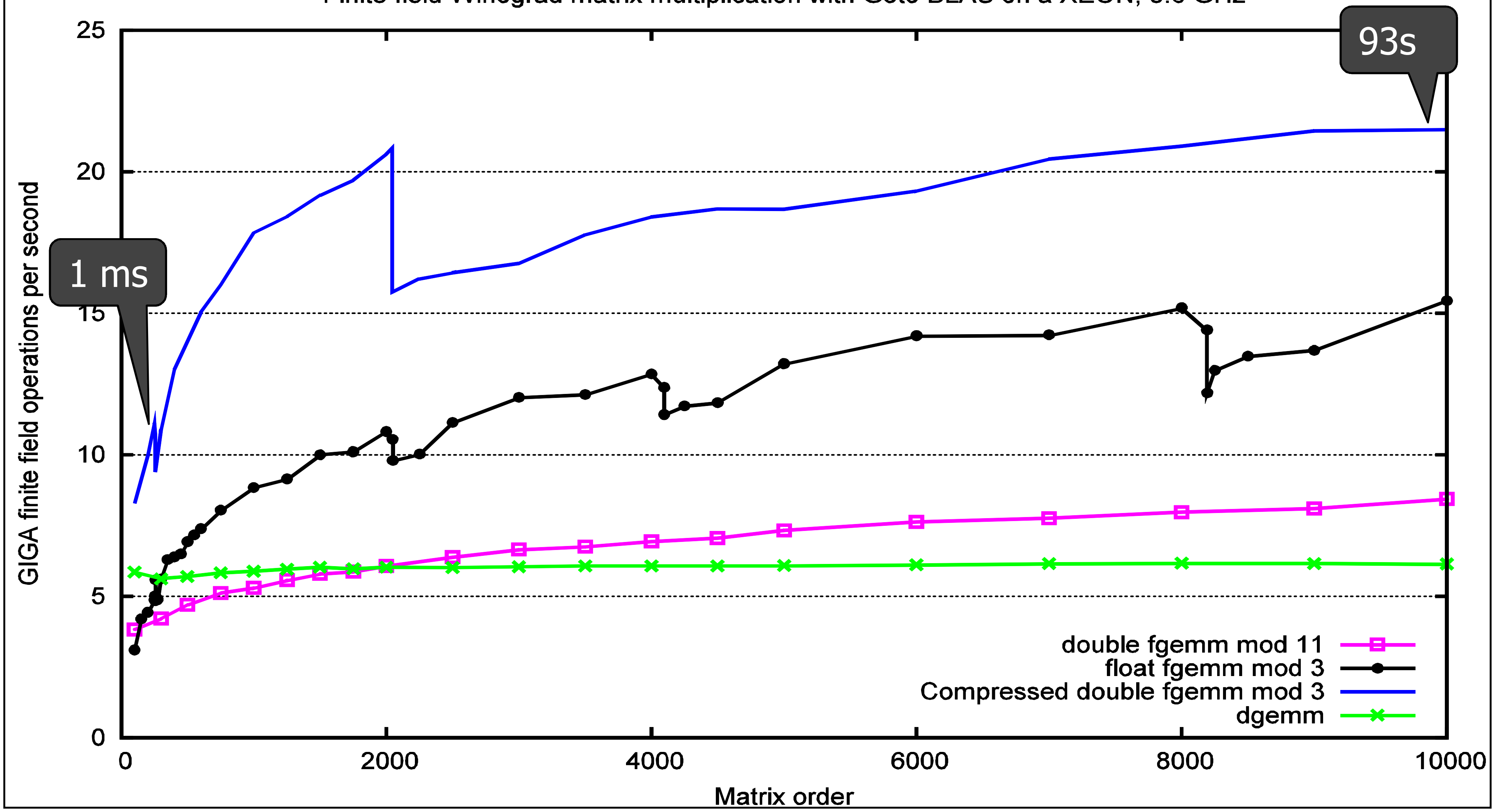


$$\sum_{i=0}^{2d} \frac{k}{d+1}(i+1)(p-1)^2 Q^i < 2^\beta$$

$$Q^{d+1} < 2^\beta$$

21.6 Gfflops on a XEON 3.6 GHz

Finite field Winograd matrix multiplication with Goto BLAS on a XEON, 3.6 GHz

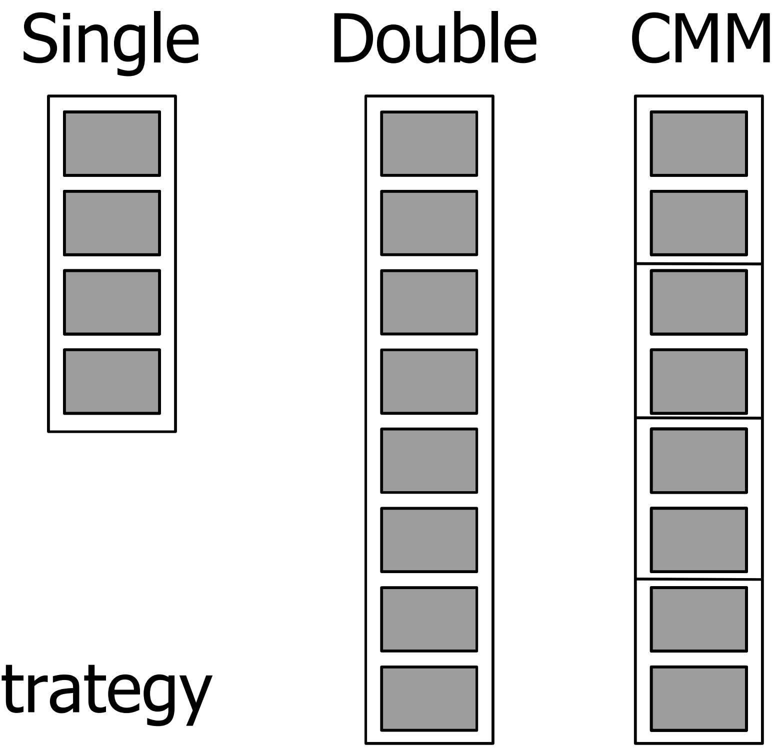


1 ms

93s

Battery of available algorithms

	Single	Double	CMM
MUL	[illegible]	8	[illegible]
RED	[illegible]	1	[illegible]



☺ LinBox 2.0 will have an adaptive strategy deciding at runtime

☺ based on (size,prime,recursive level)

⇒ This should smoothen the drops and further improve the small size cases

Further variants in the strategy

- Smaller matrices
  - Reduce the memory usage/management
- Less modular reductions
  - Speed up conversion times
- More compression
  - Less arithmetic operations

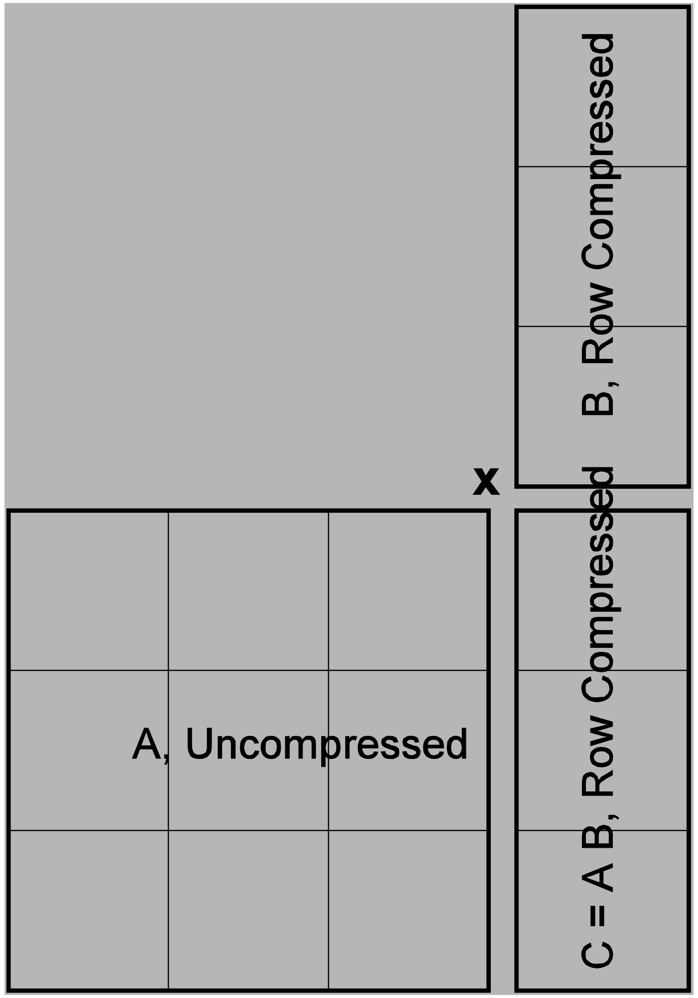
Left or Right Compression

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e + Qf \\ g + Qh \end{bmatrix} =$$

$$\begin{bmatrix} (ae + bg) + Q(af + bh) \\ (ce + dg) + Q(cf + dh) \end{bmatrix}$$

- Same bounds on Q
- But here not only the middle term needs recovery

⇒ REDQ

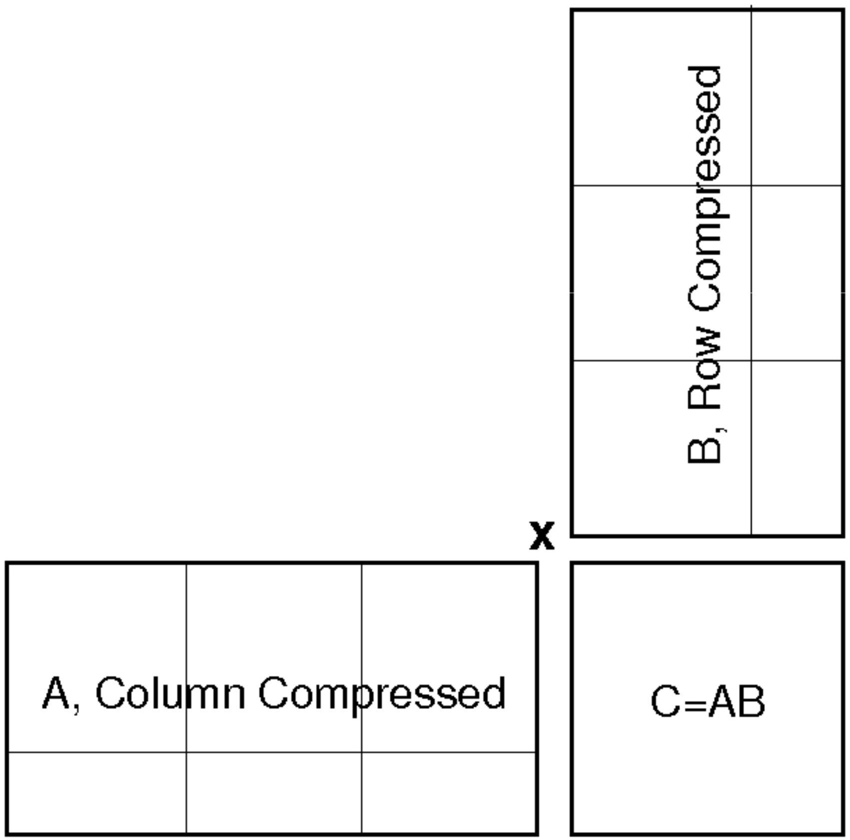


Full Compression

- Compression is squared
- Q can be R^{d+1}
- Much lower bound on Q
- Reductions are squared

$$\begin{bmatrix} a+Qc & b+Qd \end{bmatrix} \times \begin{bmatrix} e+Rf \\ g+Rh \end{bmatrix} =$$

$$\begin{bmatrix} (ae+bg)+Q(ce+dg)+R(af+bh)+QR(cf+dh) \end{bmatrix}$$

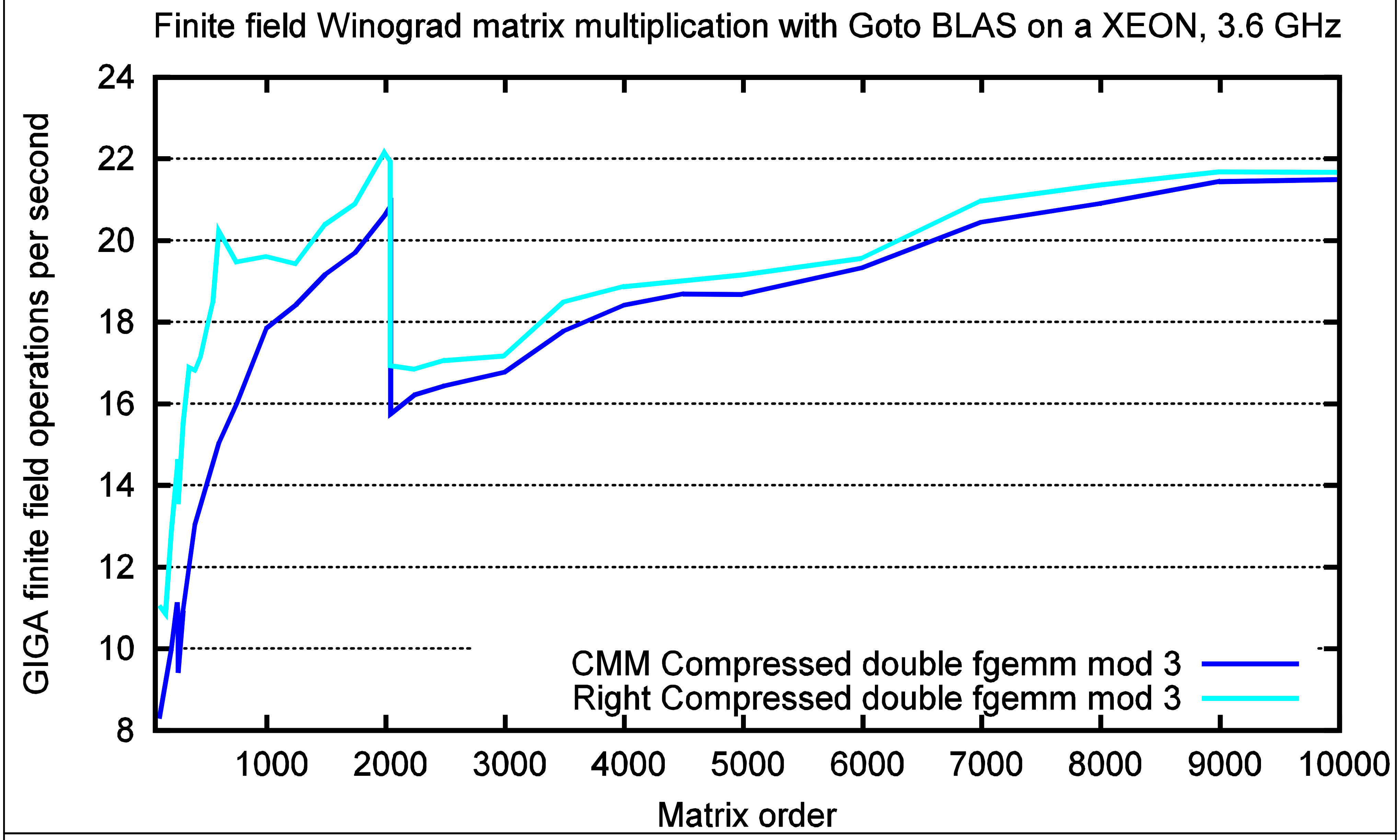


Comparison

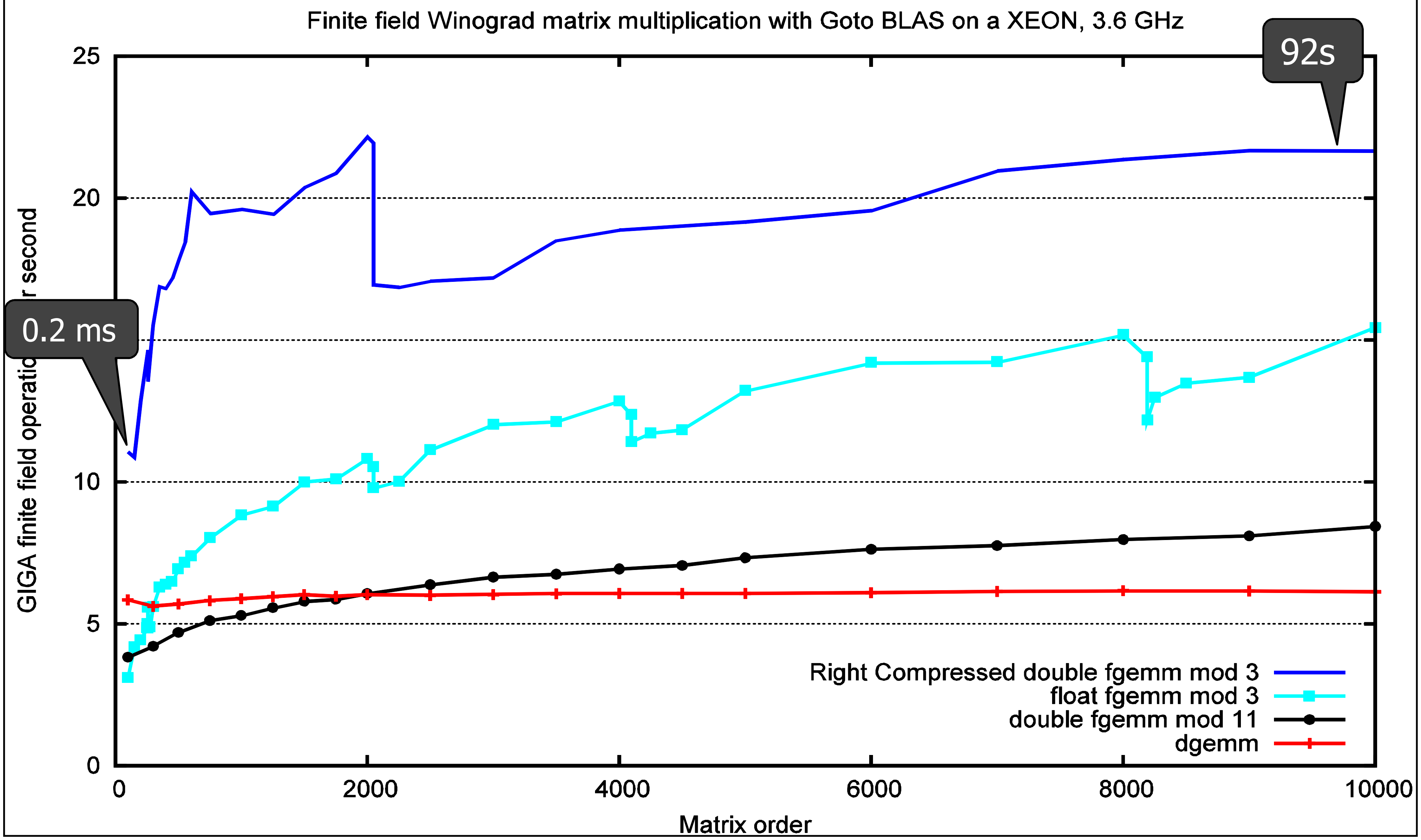
- Compression factor e = $\beta/\log_2(Q)$
  - CMM, Left or Right: d = $\lfloor e \rfloor$ - 1
  - Full: d = $\lfloor \sqrt{e} \rfloor$ - 1

Algorithm	Operations	Reductions	Conversions
CMM	$\mathcal{O}\left(mn\left(\frac{k}{\epsilon}\right)^{\omega-2}\right)$	$m \times n$ REDC	$\frac{1}{\epsilon}mn$ INIT$_\epsilon$
Right Comp.	$\mathcal{O}\left(mk\left(\frac{n}{\epsilon}\right)^{\omega-2}\right)$	$m \times \frac{n}{\epsilon}$ REDQ$_\epsilon$	$\frac{1}{\epsilon}mn$ EXTRACT$_\epsilon$
Left Comp.	$\mathcal{O}\left(nk\left(\frac{m}{\epsilon}\right)^{\omega-2}\right)$	$\frac{m}{\epsilon} \times n$ REDQ$_\epsilon$	$\frac{1}{\epsilon}mn$ EXTRACT$_\epsilon$
Full Comp.	$\mathcal{O}\left(k\left(\frac{mn}{\epsilon}\right)^{\frac{\omega-2}{2}}\right)$	$\frac{m}{\sqrt{\epsilon}} \times \frac{n}{\sqrt{\epsilon}}$ REDQ$_\epsilon$	$\frac{1}{\epsilon}mn$ INIT$_\epsilon$

Benefits of REDQ



22 Gfflops on a XEON 3.6 GHz



Compressed Arithmetic

0. Context

1. Compressed Arithmetic
   - Delayed reduction
   - Kronecker Substitution and polynomial multiplication
   - REDQ : Simultaneous Modular Reduction
   - Dot product

2. Modular Polynomial Multiplication

3. Modular Linear Algebra
   - Matrix Compression
   - REDQ with Left and Right Matrix Compression
   - Full Compression

4. Small Extension Field Linear Algebra

Word size extension field arithmetic 1/2

- We use a generator g of the invertible group of $GF(p^k)$

  e.g. $GF(9) \approx Z/3Z[X] / (X^2+X-1) = \{0\} \cup \{(X+1)^i, i=0..7\} = \{0,1,2,X,X+1,X+2,2X,2X+1,2X+2\}$
    - $(X+1)^0 = 1$
    - $(X+1)^1 = X+1$
    - $(X+1)^2 = X^2+2X+1 = X+2$
    - $(X+1)^3 = (X+2)(X+1) = 2\,X$
    - $(X+1)^4 = 2$
    - $(X+1)^5 = 2X+2$
    - $(X+1)^6 = 2X+1$
    - $(X+1)^7 = X$
    - $(X+1)^8 = 1$

Word size extension field arithmetic

- Pre-compute 3 tables
  1) Correspondence between x and i: $t_1[x] = i$, s.t. $x = g^i$
  2) Correspondence between i and x: $t_2[i] = x$, s.t. $x = g^i$
  3) « *Zech logarithm* » table: $t_3[i] = j$, s.t. $1+g^i = g^j$

- Perform operations only on the indices
  - ☺ No system division (can be 10 times slower than other arithmetic operations)
  - ⇒ Polynomial operations of degree k replaced by 2 or 3 integer operations and sometimes a table lookup
    - 0 and 1 have special values, for instance 0 and p^k-1
    - $a \times x$: $(g^i \times g^j) = g^{i+j \pm(p\textasciicircum k-1)}$
    - $x + y = g^j + g^k = g^k \times (1+g^{j-k})$

Linear Algebra over small extension fields

- Polynomials as table indexes ?
  - ☺ Kronecker substitution (p-adic transform) replaces the indeterminate by p (to minimize table size) to get a bijection
- Calling SSE, numerical BLAS routines can be 2 or 4 times faster than integer routines
- Polynomials as numerical values ?
  - + Kronecker substitution (q-adic transform) replaces the indeterminate by $q>n(p-1)^2$ (to be able to perform the linear algebra operations on the coefficients without overlapping)
  - + Delayed reduction
  - $\Rightarrow$ Works as long as $\sum (\sum a_i b_j) q^{i+j} < q^{2k-1} < 2^{53}$

Improve the q-adic algorithm

```
Algorithm 3 Polynomial dotproduct by DQT        [D., Gautier, Pernet 2002]
Input: Two vectors $v_1$ and $v_2$ in $(Z/pZ[X]/P_k)^n$ of degree less than $k$.
Input: a sufficiently large integer $q$.
Output: $R \in GF(p^k)$, with $R = v_1^T.v_2$.
    Polynomial to $q-$adic conversion
 1: Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point vectors of the evaluations at $q$
    of the elements of $v_1$
    Numerical computation (or BLAS call)
 2: Compute $\tilde{r} = \widetilde{v_1}^T.\widetilde{v_2}$
    Building the solution (can be 2k divisions)
 3: $\tilde{r} = \sum_{i=0}^{2k-2} \widetilde{\mu_i} q^i$. {Using
 4: For each $i$, set $\mu_i = \widetilde{\mu_i}$
 5: set $R = \sum_{i=0}^{2k-2} \mu_i X^i$ m
```

1: **Table lookup**

3: **REDQ simultaneous reduction**

4: **REDQ table lookup**

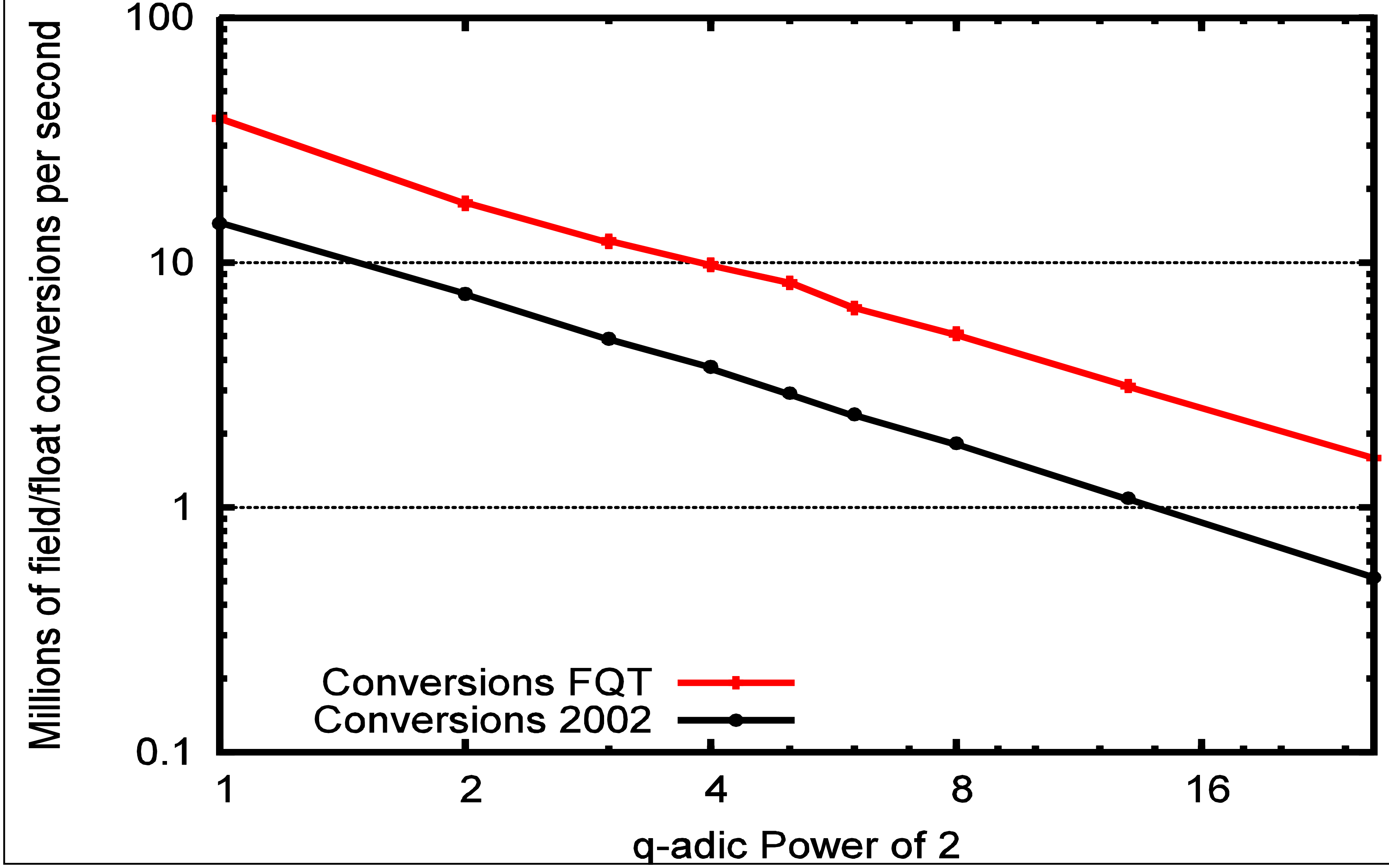
Q-adic transform revisited

```
Algorithm 4 Dot product over Galois fields via FQT
Input: a field $\mathtt{GF}(p^k)$ with elements represented as exponents of a generator of the field.
Input: Two vectors $v_1$ and $v_2$ of elements of $\mathtt{GF}(p^k)$.
Input: a sufficiently large integer $q$.
Output: $R \in \mathtt{GF}(p^k)$, with $R = v_1^T.v_2$.
    Tabulated $q-$adic conversion (1 table)
    Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point evaluations at $q$ of the elements of $v_1$ and $v_2$.
    The floating point computation
    Compute $\tilde{r} = \widetilde{v_1}^T \widetilde{v_2}$;
    Delayed reduction compression
    $[u_0, \dots u_{2k-2}] = REDQ\_COMP(\tilde{r})$
    Tabulated (2 tables) radix conversion to exponents of the generator
    Set $L = REDQ\_CORR([u_0, \dots, u_{k-1}])$              {representation of $\sum_{i=0}^{k-2} \mu_i X^i$}
    Set $H = REDQ\_CORRvariant([u_{k-1}, \dots, u_{2k-2}])$   {$H$ is $X^{k-1} \times \sum_{i=k-1}^{2k-2} \mu_i X^{i-k+1}$}
    Reduction in the field
    Return $R = H + L \in \mathtt{GF}(p^k)$;
```

Q-adic transform revisited

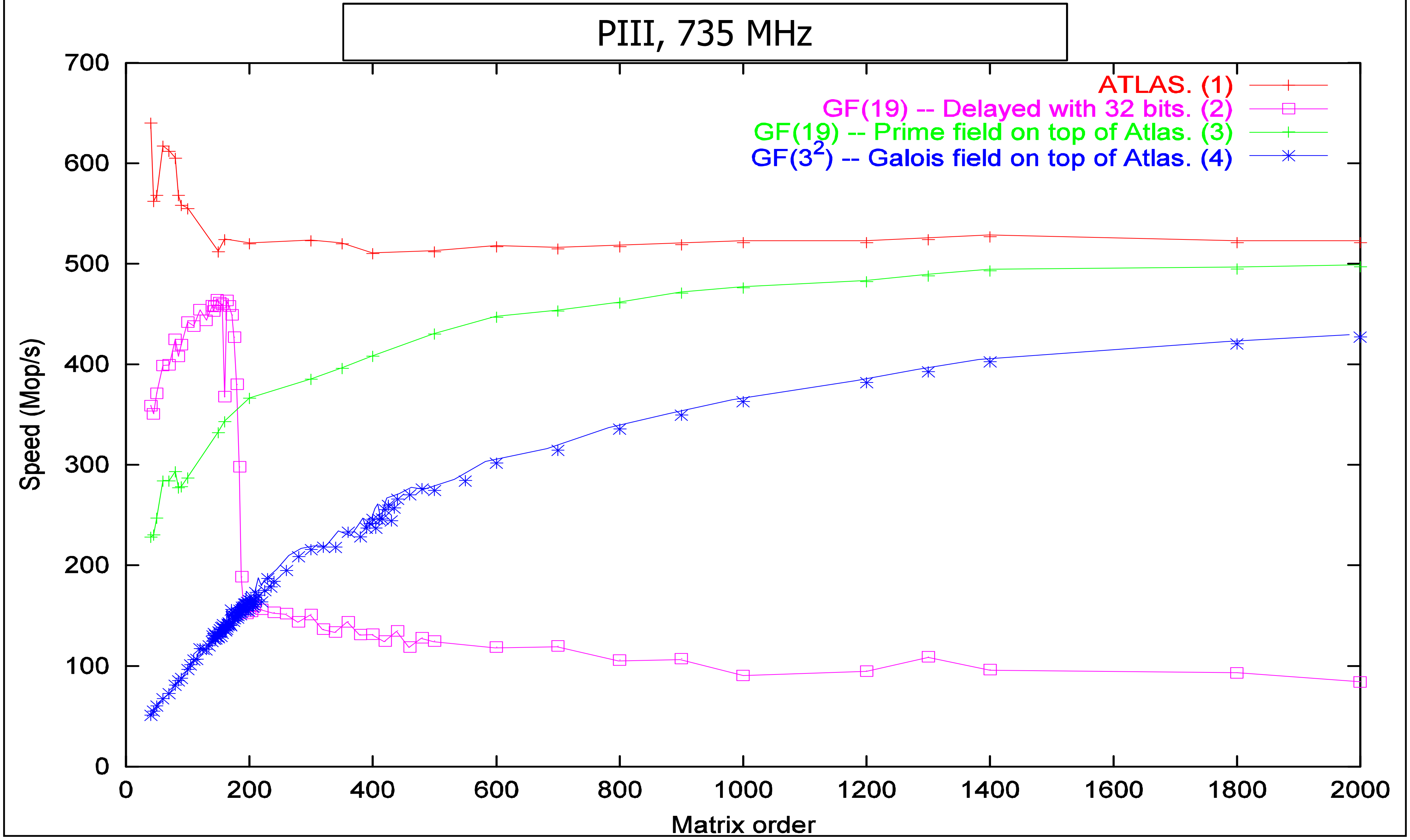
	Alg. 3	Alg. 4
Memory	$3p^k$	$4p^k + 2^{1+k\lceil\log_2 p\rceil}$
Axpy	0	k
Div	$2k-1$	0
Table	0	3
Red	$\geq 5k$	1

Conversions: FQT vs 2002 algorithm



Finite Field Linear Algebra

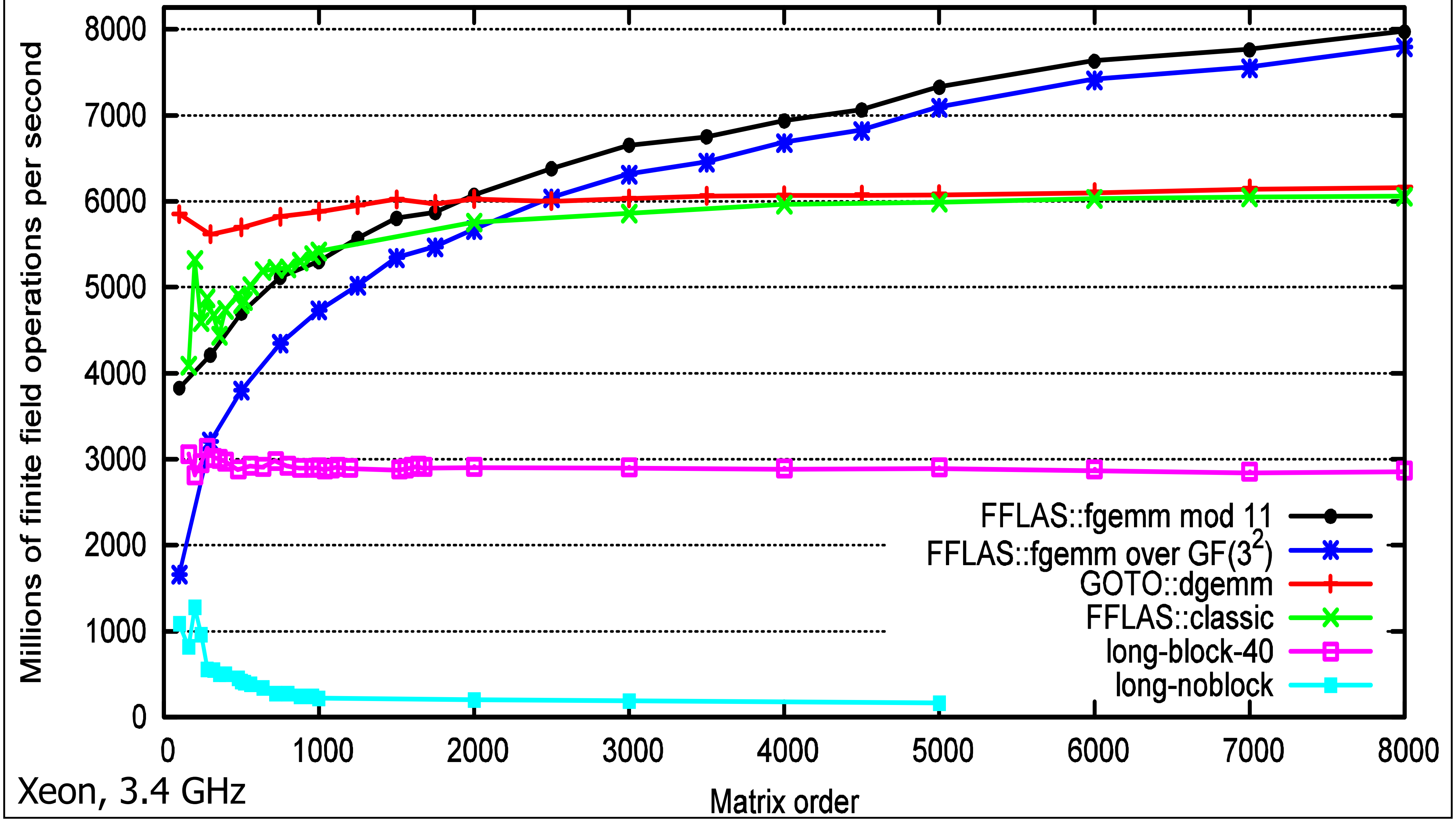
[D., Gautier, Pernet, 2002]



fgemm today

[D,Giorgi,Pernet 2009]

Hz x 4.6 ; BLAS x 12 ; GF(9) x 19.2



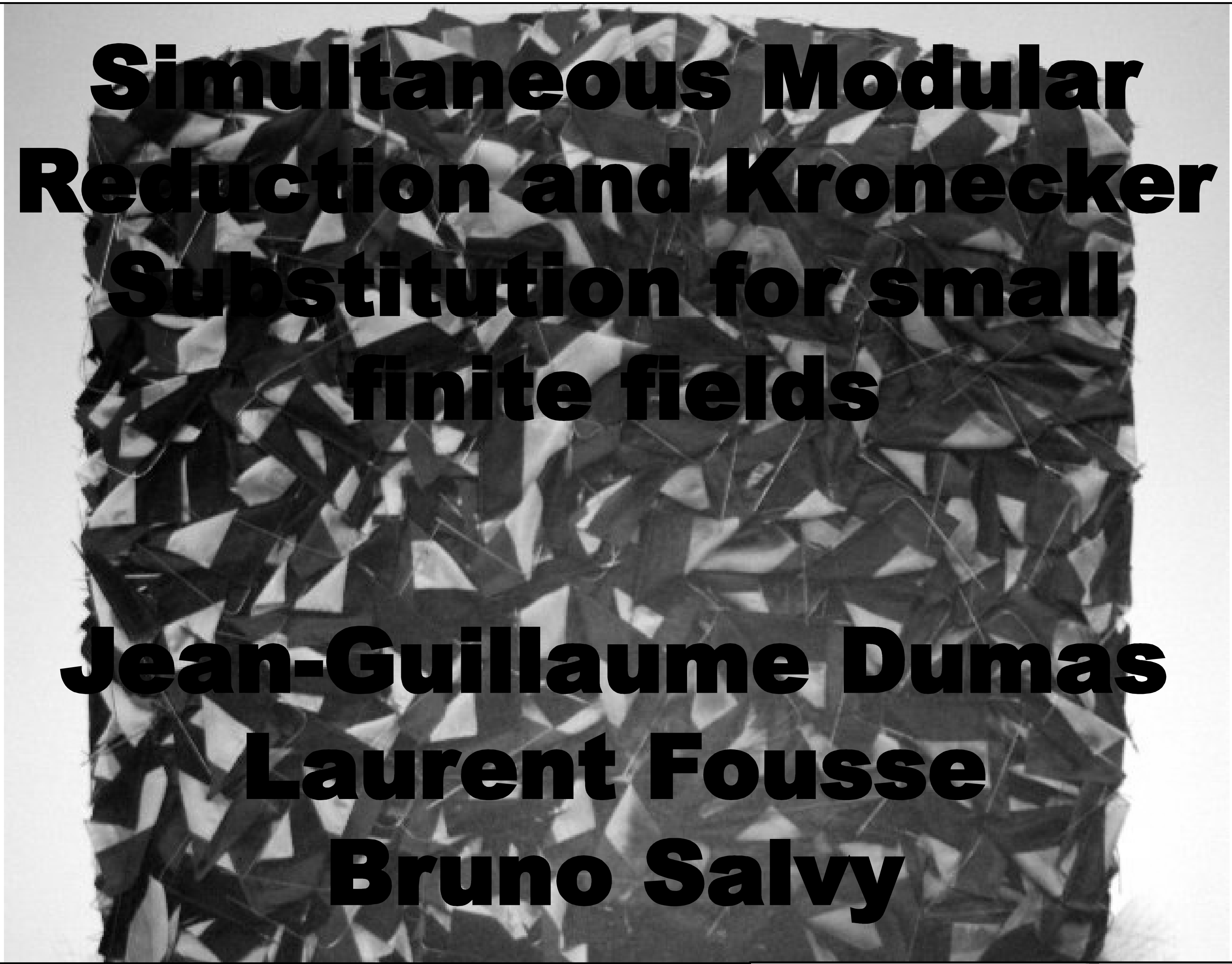
Conclusion

- Compressed arithmetic gains constant factors
  - 64bits: Degree 3/4/5 Polynomial multiplication at cost 1
  - 64bits: Size 3/4/5 dotproduct at cost 1
  - Some larger precision arithmetic could be used …
- The FFLAS paradigm is to convert towards a representation where cache/SSE/multicore efficient routines exist
  - Integer SSE (2009?) will extend the mantissa from 53 to 64 bits
  - Extended BLAS *[Demmel et al.]* or Complex BLAS could give 128 bits …

Perspectives

- Implementations of Full compression
- Explore other choices of q
- Automatic recursive cutting:
  - e.g. : n=2048 can use compression factor of 4 where n=2049 can use only a factor of 3
  - Alternative: compute multiplications of size 1024 with compression factor of 4 and the highest recursive level within the uncompressed field

$\Rightarrow$ smoothen the drops at the change of compression factor



Simultaneous Modular Reduction and Kronecker Substitution for small finite fields

Jean-Guillaume Dumas
Laurent Fousse
Bruno Salvy

Grenoble University
Laboratoire Jean Kuntzmann
Applied Mathematics and Computer Science Department

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE
INRIA
centre de recherche PARIS - ROCQUENCOURT