Toward High Performance Matrix Multiplication for Exact Computation

Pascal Giorgi

Joint work with Romain Lebreton (U. Waterloo)
Funded by the French ANR project HPAC

Séminaire CASYS - LJK, April 2014
Motivations

- Matrix multiplication plays a central role in computer algebra. \\textit{algebraic complexity of } O(n^\omega) \text{ with } \omega < 2.3727 \text{ [Williams 2011]}

- Modern processors provide many levels of parallelism. \\
\textit{superscalar, SIMD units, multiple cores}

High performance matrix multiplication

- numerical computing = classic algorithm + hardware arithmetic
- exact computing $\neq$ numerical computing
  - algebraic algorithm is not the most efficient ($\neq$ complexity model)
  - arithmetic is not directly in the hardware (e.g. $\mathbb{Z}$, $F_q$, $\mathbb{Z}[x]$, $\mathbb{Q}[x, y, z]$).
Motivation: Superscalar processor with SIMD

Hierarchical memory:
- L1 cache: 32kB - 4 cycles
- L2 cache: 256kB - 12 cycles
- L3 cache: 8MB - 36 cycles
- RAM: 32GB - 36 cycles + 57 ns
Motivations: practical algorithms

**High performance algorithms (rule of thumb)**
- best asymptotic complexity is not always faster: *constants matter*
- better arithmetic count is not always faster: *caches matter*
- process multiple data at the same time: *vectorization*
- fine/coarse grain task parallelism matter: *multicore parallelism*
Motivations: practical algorithms

High performance algorithms (rule of thumb)

- best asymptotic complexity is not always faster: constants matter
- better arithmetic count is not always faster: caches matter
- process multiple data at the same time: vectorization
- fine/coarse grain task parallelism matter: multicore parallelism

Our goal: try to incorporate these rules into exact matrix multiplications
Outline

1. Matrix multiplication with small integers
2. Matrix multiplication with multi-precision integers
3. Matrix multiplication with polynomials
Outline

1. Matrix multiplication with small integers
2. Matrix multiplication with multi-precision integers
3. Matrix multiplication with polynomials
Matrix multiplication with small integers

This corresponds to the case where each integer result holds in one processor register:

\[ A, B \in \mathbb{Z}^{n \times n} \text{ such that } ||AB||_\infty < 2^s \]

where \( s \) is the register size.

**Main interests**
- ring isomorphism:
  \[ \rightarrow \text{ computation over } \mathbb{Z}/p\mathbb{Z} \text{ is congruent to } \mathbb{Z}/2^s\mathbb{Z} \text{ when } p(n - 1)^2 < 2^s. \]
- its a building block for matrix multiplication with larger integers
Matrix multiplication with small integers

Two possibilities for hardware support:
- use floating point mantissa, i.e. \( s = 2^{53} \),
- use native integer, i.e. \( s = 2^{64} \).

### Using floating point

- Historically, the first approach in computer algebra [Dumas, Gautier, Pernet 2002]
  - Out of the box performance from optimized BLAS
  - ✗ Handle matrix with entries \(< 2^{26}\)

### Using native integers

- ✓ Apply same optimizations as BLAS libraries [Goto, Van De Geijn 2008]
- ✓ Handle matrix with entries \(< 2^{32}\)
Matrix multiplication with small integers

<table>
<thead>
<tr>
<th></th>
<th>floating point</th>
<th>integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem (2008)</td>
<td>1 mul+1 add</td>
<td>1 mul+2 add</td>
</tr>
<tr>
<td>Sandy Bridge (2011)</td>
<td>1 mul+1 add</td>
<td>1 mul+2 add</td>
</tr>
<tr>
<td>Haswell (2013)</td>
<td>2 FMA</td>
<td>1 mul+2 add</td>
</tr>
</tbody>
</table>

# vector operations per cycle (pipelined)

<table>
<thead>
<tr>
<th>Matrix dimensions</th>
<th>OpenBlas: double - SSE</th>
<th>OpenBlas: double - AVX</th>
<th>OpenBlas: double - AVX2</th>
<th>Our code: int - SSE</th>
<th>Our code: int - AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

benchmark on Intel i7-4960HQ @ 2.60GHz
Matrix multiplication with small integers

Matrix multiplication modulo a small integer

Let $p$ such that $(p - 1)^2 \times n < 2^{53}$

1. perform the multiplication in $\mathbb{Z}$ using BLAS
2. reduce the result modulo $p$

Benchmark on Intel i7-4960HQ @ 2.60GHz
Outline

1. Matrix multiplication with small integers
2. Matrix multiplication with multi-precision integers
3. Matrix multiplication with polynomials
Matrix multiplication with multi-precision integers

**Direct approach**

Let $M(k)$ be the bit complexity of $k$-bit integers multiplication and $A, B \in \mathbb{Z}^{n \times n}$ such that $||A||_{\infty}, ||B||_{\infty} \in O(2^k)$.

Computing $AB$ using direct algorithm costs $n^\omega M(k)$ bit operations.

- not best possible complexity, i.e. $M(k)$ is super-linear
- not efficient in practice

Remark:
Use evaluation/interpolation technique for better performances!!!
Multi-modular matrix multiplication

Multi-modular approach

\[ ||AB||_\infty < M = \prod_{i=1}^{k} m_i, \quad \text{with primes } m_i \in O(1) \]

then \( AB \) can be reconstructed with the CRT from \((AB) \mod m_i\).

1. for each \( m_i \) compute \( A_i = A \mod m_i \) and \( B_i = B \mod m_i \)
2. for each \( m_i \) compute \( C_i = A_i B_i \mod m_i \)
3. reconstruct \( C = AB \) from \((C_1, \ldots, C_k)\)

Bit complexity:

\[ O(n^\omega k + n^2 R(k)) \]

where \( R(k) \) is the cost of reduction/reconstruction
Multi-modular matrix multiplication

Multi-modular approach

\[ \|AB\|_\infty < M = \prod_{i=1}^{k} m_i, \quad \text{with primes } m_i \in O(1) \]

then \( AB \) can be reconstructed with the CRT from \((AB) \mod m_i\).

1. For each \( m_i \) compute \( A_i = A \mod m_i \) and \( B_i = B \mod m_i \)
2. For each \( m_i \) compute \( C_i = A_iB_i \mod m_i \)
3. Reconstruct \( C = AB \) from \((C_1, \ldots, C_k)\)

Bit complexity :
\[ O(n^\omega k + n^2 R(k)) \] where \( R(k) \) is the cost of reduction/reconstruction

- \( R(k) = O(M(k) \log(k)) \) using divide and conquer strategy
- \( R(k) = O(k^2) \) using naive approach
Multi-modular matrix multiplication

Improving naive approach with linear algebra

- Reduction/reconstruction of $n^2$ data corresponds to matrix multiplication
- ✓ Improve the bit complexity from $O(n^2 k^2)$ to $O(n^2 k^{\omega-1})$
- ✓ Benefit from optimized matrix multiplication, i.e. SIMD

Remark:
A similar approach has been used by [Doliskani, Schost 2010] in a non-distributed code.
Multi-modular reductions of an integer matrix

Let us assume \( M = \prod_{i=1}^{k} m_i < \beta^k \) with \( m_i < \beta \).

Multi-reduction of a single entry

Let \( a = a_0 + a_1\beta + \ldots a_{k-1}\beta^{k-1} \) be a value to reduce mod \( m_i \) then

\[
\begin{bmatrix}
|a|_{m_1} \\
\vdots \\
|a|_{m_k}
\end{bmatrix} = \begin{bmatrix} 1 & |\beta|_{m_1} & \ldots & |\beta^{k-1}|_{m_1} \\
\vdots & \vdots & \ddots & \vdots \\
1 & |\beta|_{m_k} & \ldots & |\beta^{k-1}|_{m_k}
\end{bmatrix} \times \begin{bmatrix} a_0 \\
\vdots \\
a_{k-1}
\end{bmatrix} - Q \times \begin{bmatrix} m_1 \\
\vdots \\
m_k
\end{bmatrix}
\]

with \( ||Q||_\infty < k\beta^2 \)

Lemma: if \( k\beta^2 \in O(1) \) than the reduction of \( n^2 \) integers modulo the \( m_i \)'s costs \( O(n^2 k^{\omega-1}) + O(n^2 k) \) bit operations.
Multi-modular reconstruction of an integer matrix

Let us assume \( M = \prod_{i=1}^{k} m_i < \beta^k \) with \( m_i < \beta \) and \( M_i = M/m_i \)

\[
\text{CRT formulae: } a = \left( \sum_{i=1}^{k} |a|_{m_i} \cdot M_i |M_i^{-1}|_{m_i} \right) \mod M
\]

Reconstruction of a single entry

Let \( M_i |M_i^{-1}|_{m_i} = \alpha_{0}^{(i)} + \alpha_{1}^{(i)} \beta + \ldots + \alpha_{k-1}^{(i)} \beta^{k-1} \) be the CRT constants, then

\[
\begin{bmatrix}
a_0 \\
\vdots \\
a_{k-1}
\end{bmatrix} =
\begin{bmatrix}
\alpha_{0}^{(1)} & \ldots & \alpha_{k-1}^{(1)} \\
\vdots & \ddots & \vdots \\
\alpha_{0}^{(k)} & \ldots & \alpha_{k-1}^{(k)}
\end{bmatrix}
\times
\begin{bmatrix}
|a|_{m_1} \\
\vdots \\
|a|_{m_k}
\end{bmatrix}
\]

with \( a_i < k \beta^2 \) and \( a = a_0 + \ldots + a_{k-1} \beta^{k-1} \mod M \) the CRT solution.

Lemma: if \( k \beta^2 \in O(1) \) than the reconstruction of \( n^2 \) integers from their images modulo the \( m_i \)’s costs \( O(n^2 k^{\omega-1}) + O^\sim(n^2 k) \) bit operations.
Matrix multiplication with multi-precision integers

Implementation of multi-modular approach

- choose $\beta = 2^{16}$ to optimize $\beta$-adic conversions
- choose $m_i$ s.t. $n\beta m_i < 2^{53}$ and use BLAS `dgemm`
- use a linear storage for multi-modular matrices

Compare sequential performances with:

- FLINT library\(^1\) : uses divide and conquer
- Mathemagix library\(^2\) : uses divide and conquer
- Doliskani’s code\(^3\) : uses `dgemm` for reductions only

\(^1\)www.flintlib.org
\(^2\)www.mathemagix.org
\(^3\)courtesy of J. Doliskani
Matrix multiplication with multi-precision integers

Integer matrix multiplication
(matrix dim = 32)

- Flint
- Mathemagix
- Doliskani’s code
- Our code

benchmark on Intel Xeon-2620 @ 2.0GHz
Matrix multiplication with multi-precision integers

Integer matrix multiplication
(matrix dim = 128)

- Flint
- Mathemagix
- Doliskani’s code
- Our code

benchmark on Intel Xeon-2620 @ 2.0GHz
Matrix multiplication with multi-precision integers

Integer matrix multiplication
(matrix dim = 512)

Time in seconds vs. Entry bitsize for different integer matrix multiplication libraries:
- Flint
- Mathemagix
- Doliskani’s code
- Our code

Benchmark on Intel Xeon-2620 @ 2.0GHz
Matrix multiplication with multi-precision integers

Integer matrix multiplication  
(matrix dim = 1024)

benchmark on Intel Xeon-2620 @ 2.0GHz
Parallel multi-modular matrix multiplication

- for $i = 1 \ldots k$ compute $A_i = A \mod m_i$ and $B_i = B \mod m_i$

- reconstruct $C = AB$ from $(C_1, \ldots, C_k)$

Parallelization of multi-modular reduction/reconstruction

Each thread reduces (resp. reconstructs) a chunk of the given matrix.

<table>
<thead>
<tr>
<th>thread 0</th>
<th>thread 1</th>
<th>thread 2</th>
<th>thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$A_0 = A \mod m_0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_1 = A \mod m_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_2 = A \mod m_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_3 = A \mod m_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_4 = A \mod m_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_5 = A \mod m_5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A_6 = A \mod m_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallel multi-modular matrix multiplication

for $i = 1 \ldots k$ compute $C_i = A_i B_i \mod m_i$

<table>
<thead>
<tr>
<th>thread 0</th>
<th>$C_0 = A_0 B_0 \mod m_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 1</td>
<td>$C_1 = A_1 B_1 \mod m_1$</td>
</tr>
<tr>
<td>thread 2</td>
<td>$C_2 = A_2 B_2 \mod m_2$</td>
</tr>
<tr>
<td>thread 3</td>
<td>$C_3 = A_3 B_3 \mod m_3$</td>
</tr>
<tr>
<td></td>
<td>$C_4 = A_4 B_4 \mod m_4$</td>
</tr>
<tr>
<td></td>
<td>$C_5 = A_5 B_5 \mod m_5$</td>
</tr>
<tr>
<td></td>
<td>$C_6 = A_6 B_6 \mod m_6$</td>
</tr>
</tbody>
</table>
Parallel Matrix multiplication with multi-precision integers

- based on OpenMP task
- CPU affinity (hwloc-bind), allocator (tcmalloc)
- still under progress for better memory strategy!!!
Outline

1. Matrix multiplication with small integers
2. Matrix multiplication with multi-precision integers
3. Matrix multiplication with polynomials
Matrix multiplication over $F_p[x]$

We consider the "easiest" case:

$$A, B \in F_p[x]^{n \times n} \text{ such that } \deg(AB) < k = 2^t$$

- $p$ is a Fourier prime, i.e. $p = 2^t q + 1$
- $p$ is such that $n(p - 1)^2 < 2^{53}$

**Complexity**

$O(n^{\omega} k + n^2 k \log(k))$ op. in $F_p$ using evaluation/interpolation with FFT
Matrix multiplication over $\mathbb{F}_p[x]$

We consider the "easiest" case:

$$A, B \in \mathbb{F}_p[x]^{n \times n} \text{ such that } \deg(AB) < k = 2^t$$

- $p$ is a Fourier prime, i.e. $p = 2^t q + 1$
- $p$ is such that $n(p - 1)^2 < 2^{53}$

**Complexity**

$O(n^\omega k + n^2 k \log(k))$ op. in $\mathbb{F}_p$ using evaluation/interpolation with FFT

**Remark:**

using Vandermonde matrix on can get a similar approach as for integers, i.e. $O(n^\omega k + n^2 k^{\omega-1})$
Matrix multiplication over $\mathbb{F}_p[x]$

**Evaluation/Interpolation scheme**

Let $\theta$ a primitive $k$th root of unity in $\mathbb{F}_p$.

1. for $i = 1 \ldots k$ compute $A_i = A(\theta^{i-1})$ and $B_i = B(\theta^{i-1})$
2. for $i = 1 \ldots k$ compute $C_i = A_i B_i \in \mathbb{F}_p$
3. interpolate $C = AB$ from $(C_1, \ldots, C_k)$

- steps 1 and 3: $O(n^2)$ call to $\text{FFT}_k$ over $\mathbb{F}_p[x]$
- step 2: $k$ matrix multiplications modulo a small prime $p$
FFT with SIMD over $F_p$

Butterfly operation modulo $p$

compute $X + Y \mod p$ and $(X - Y)\theta^{2^i} \mod p$.

- Barret’s modular multiplication with a constant (NTL)
- calculate into $[0, 2p)$ to remove two conditionals [Harvey 2014]

Let $X, Y \in [0, 2p), W \in [0, p), p < \beta/4$ and $W' = \lceil W\beta/p \rceil$.

Algorithm: Butterfly($X,Y,W,W',p$)

1: $X' := X + Y \mod 2p$
2: $T := X - Y + 2p$
3: $Q := \lceil W'T/\beta \rceil$ 1 high short product
4: $Y' := (WT - Qp) \mod \beta$ 2 low short products
5: return $(X', Y')$
FFT with SIMD over $F_p$

**Butterfly operation modulo p**

compute $X + Y \mod p$ and $(X - Y)\theta^{2^i} \mod p$.

- Barret's modular multiplication with a constant (NTL)
- calculate into $[0, 2p)$ to remove two conditionals [Harvey 2014]

Let $X, Y \in [0, 2p), W \in [0, p), p < \beta/4$ and $W' = \lceil W\beta/p \rceil$.

**Algorithm:** Butterfly($X, Y, W, W', p$)

1. $X' := X + Y \mod 2p$
2. $T := X - Y + 2p$
3. $Q := \lceil W' T/\beta \rceil$
4. $Y' := (WT - Qp) \mod \beta$
5. return $(X', Y')$

- ✔️ SSE/AVX provide 16 or 32-bits low short product
- ✗ no high short product available (use full product)
Matrix multiplication over $F_p[x]$

**Implementation**

- radix-4 FFT with 128-bits SSE (29 bits primes)
- BLAS-based matrix multiplication over $F_p$ [FFLAS-FFPACK library]

---

**Polynomial Matrix Multiplication Performances**

<table>
<thead>
<tr>
<th>Matrix with dimension (n) and degree (deg)</th>
<th>FLINT</th>
<th>MMX</th>
<th>our code</th>
</tr>
</thead>
<tbody>
<tr>
<td>deg=1024 n=16</td>
<td>0.02</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=2048 n=16</td>
<td>0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=4096 n=16</td>
<td>0.06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=8192 n=16</td>
<td>0.13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=16</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=256 n=16</td>
<td>0.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=128</td>
<td>1.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=256 n=128</td>
<td>2.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=512</td>
<td>4.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=128 n=128</td>
<td>8.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=512</td>
<td>16.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=256 n=512</td>
<td>32.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=512</td>
<td>64.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=256 n=512</td>
<td>128.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=1024</td>
<td>256.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=256 n=1024</td>
<td>512.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deg=512 n=2048</td>
<td>1024.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*benchmark on Intel Xeon-2620 @ 2.0GHz*
Matrix multiplication over $\mathbb{Z}[x]$

$A, B \in \mathbb{Z}[x]^{n \times n}$ such that $\deg(AB) < d$ and $\|(AB)_i\|_\infty < k$

**Complexity**

- $O^{\sim}(n^\omega d \log(d) \log(k))$ bit op. using Kronecker substitution
- $O(n^\omega d \log(k) + n^2 d \log(d) \log(k))$ bit op. using CRT+FFT

**Remark:**

if the result’s degree and bitsize are not too large, CRT with Fourier primes might suffice.
Matrix multiplication over $\mathbb{Z}[x]$

**Implementation**

- use CRT with Fourier primes
- re-use multi-modular reduction/reconstruction with linear algebra
- re-use multiplication in $F_p[x]$

### Benchmark Results

<table>
<thead>
<tr>
<th>$\log(k)$</th>
<th>$n=16$; $\text{deg}=8192$</th>
<th>$n=128$; $\text{deg}=512$</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

*benchmark on Intel Xeon-2620 @ 2.0GHz*
Parallel Matrix multiplication over $\mathbb{Z}[x]$

Very first attempt (work still in progress)
- parallel CRT with linear algebra (same code as in $\mathbb{Z}$ case)
- perform each multiplication over $\mathbb{F}_p[x]$ in parallel
- some part of the code still sequential

<table>
<thead>
<tr>
<th>$n$</th>
<th>$d$</th>
<th>$\log(k)$</th>
<th>6 cores</th>
<th>12 cores</th>
<th>time seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1024</td>
<td>600</td>
<td>$\times 3.52$</td>
<td>$\times 4.88$</td>
<td>61.1s</td>
</tr>
<tr>
<td>32</td>
<td>4096</td>
<td>600</td>
<td>$\times 3.68$</td>
<td>$\times 5.02$</td>
<td>64.4s</td>
</tr>
<tr>
<td>32</td>
<td>2048</td>
<td>1024</td>
<td>$\times 3.95$</td>
<td>$\times 5.73$</td>
<td>54.5s</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>1024</td>
<td>$\times 3.76$</td>
<td>$\times 5.55$</td>
<td>53.9s</td>
</tr>
</tbody>
</table>
Generic handler class for Polynomial Matrix

```cpp
template<
sizet type, sizet storage, class Field>
class PolynomialMatrix;
```

Specialization for different memory strategy

```cpp
// Matrix of polynomials
template<class _Field>
class PolynomialMatrix<PMTyp::polfirst, PMStorage::plain, _Field>;

// Polynomial of matrices
template<class _Field>
class PolynomialMatrix<PMTyp::matfirst, PMStorage::plain, _Field>;

// Polynomial of matrices (partial view on monomials)
template<class _Field>
class PolynomialMatrix<PMTyp::matfirst, PMStorage::view, _Field>;
```
Conclusion

High performance tools for exact linear algebra:

- matrix multiplication through floating points
- multi-dimensional CRT
- FFT for polynomial over wordsize prime fields
- adaptative matrix representation

We provide in the LinBox library (www.linalg.org)

- efficient sequential/parallel matrix multiplication over $\mathbb{Z}$
- efficient sequential matrix multiplication over $F_p[x]$ and $\mathbb{Z}[x]$