Efficient Modular Matrix Multiplication on GPU for Polynomial System Solving

Jérémy Berthomieu, Stef Graillat, **Dimitri Lesnoff**, Theo Mary

LIP6 — Sorbonne Université – CNRS

ISSAC, $24 - 27$ July 2023, Tromsø

Motivation: Polynomial System Solving

0-dim Polynomial System

$\mathbb{K} = \mathbb{F}_p$, p prime,
$x_1, \ldots, x_n$ unknowns,
$f_1, \ldots, f_n$ polynomials

$$\begin{cases} f_1(x_1, \ldots, x_n) & = 0 \\ \vdots & = \vdots \\ f_n(x_1, \ldots, x_n) & = 0 \end{cases}$$

Motivation: Polynomial System Solving

0-dim Polynomial System

$\mathbb{K} = \mathbb{F}_p$, p prime,
$x_1, \ldots, x_n$ unknowns,
$f_1, \ldots, f_n$ polynomials

$$\begin{cases} f_1(x_1, \ldots, x_n) &= 0 \\ \quad\vdots &= \vdots \\ f_n(x_1, \ldots, x_n) &= 0 \end{cases}$$

$\longrightarrow$

Shape Position

$$\begin{cases} g_n(x_n) &= 0 \\ x_{n-1} &= g_{n-1}(x_n) \\ \vdots &= \vdots \\ x_1 &= g_1(x_n) \end{cases}$$

Motivation: Polynomial System Solving

0-dim Polynomial System

$\mathbb{K} = \mathbb{F}_p$, p prime,
$x_1, \ldots, x_n$ unknowns,
$f_1, \ldots, f_n$ polynomials

$$\begin{cases} f_1(x_1, \ldots, x_n) &= 0 \\ \vdots &= \vdots \\ f_n(x_1, \ldots, x_n) &= 0 \end{cases}$$

$\longrightarrow$

Shape Position

$$\begin{cases} g_n(x_n) &= 0 \\ x_{n-1} &= g_{n-1}(x_n) \\ \vdots &= \vdots \\ x_1 &= g_1(x_n) \end{cases}$$

SparseFGLM [Faugère, Mou, 2011, 2017] with Block-Wiedemann

g_n: minimal polynomial of a matrix M
M: multiplication matrix in x_n w.r.t. a DRL Gröbner basis
Bottleneck:
Matrix sequence computation: $2D/s$ matrix products of size $t \times D$ and $D \times s$
D: degree of the ideal, $t \simeq D/3$, $s = 32$

Challenges and Plan of the Talk

Challenges

Large scale matrix multiplication: $M' \cdot N, \quad M' \in \mathbb{F}_p^{t \times D}, N \in \mathbb{F}_p^{D \times s}, D \simeq 100\,000$

Field: $\mathbb{K} = \mathbb{F}_p$, p prime, $\texttt{bitsize}(p) \geq 26$ bits

Challenges and Plan of the Talk

Challenges

Large scale matrix multiplication: $M' \cdot N$, $\quad M' \in \mathbb{F}_p^{t \times D}, N \in \mathbb{F}_p^{D \times s}, D \simeq 100\,000$

Field: $\mathbb{K} = \mathbb{F}_p$, p prime, $\texttt{bitsize}(p) \geq 26$ bits

Objectives

1. Going faster: Use **GPU** parallel architecture and BLAS $\rightarrow$ `CUBLAS`, `rocBLAS`
   $\rightarrow$ matrix product over finite field with **CUBLAS** on **GPU**
2. Going further: Lift the prime limit of 26 bits while preserving efficiency
   $\rightarrow$ **Multi-word** matrix product

CPU/GPU Software and Libraries

CPU libraries

- NTL [Shoup 2002]
- FLINT [Hart, Johansson, Pancratz, 2007]
- FFLAS-FFPACK [Dumas, Giorgi, Pernet, 2008]

GPU software

- MAGMA [Steel, 2015]

CPU/GPU Software and Libraries

CPU libraries

- NTL [Shoup 2002]
- FLINT [Hart, Johansson, Pancratz, 2007]
- FFLAS-FFPACK [Dumas, Giorgi, Pernet, 2008]

GPU software

- MAGMA [Steel, 2015]

Floating-point representation between 2^8 and 2^{26} $\rightarrow$ efficient.
Integer arithmetic after prime limit, accumulation must be exact:

binary32: (`float`) 2^{24} limit $\rightarrow$ $p \leq 2887$ 😡

binary64: (`double`) 2^{53} limit $\rightarrow$ $p < 2^{26}$ 🙁

CPU/GPU Software and Libraries

CPU libraries

- NTL [Shoup 2002]
- FLINT [Hart, Johansson, Pancratz, 2007]
- FFLAS-FFPACK [Dumas, Giorgi, Pernet, 2008]

GPU software

- MAGMA [Steel, 2015]

Floating-point representation between 2^8 and 2^{26} $\rightarrow$ efficient.
Integer arithmetic after prime limit, accumulation must be exact:

binary32: (`float`) 2^{24} limit $\rightarrow$ $p \leq 2887$ 😡

binary64: (`double`) 2^{53} limit $\rightarrow$ $p < 2^{26}$ 🙁

Scarce native support for 64-bit integers on GPU $\rightarrow$ Floating-point types are a must!

Going Faster: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

Algorithm: λ-block matrix product over $\mathbb{F}_p$

Input : $A \in \mathbb{F}_p^{t\times D}$, $B \in \mathbb{F}_p^{D\times s}$, p,

$\lambda = \left\lfloor (2^{53} - p - 1)/(p-1)^2 \right\rfloor$

$\simeq 2^{53-2r}$ where $r = \texttt{bitsize}(p)$

Assumption: $a_{i,j}, b_{i,j} < 2^{26}$ stored as `binary64`

Output : $C = AB \in \mathbb{F}_p^{t\times s}$ stored as `binary64`

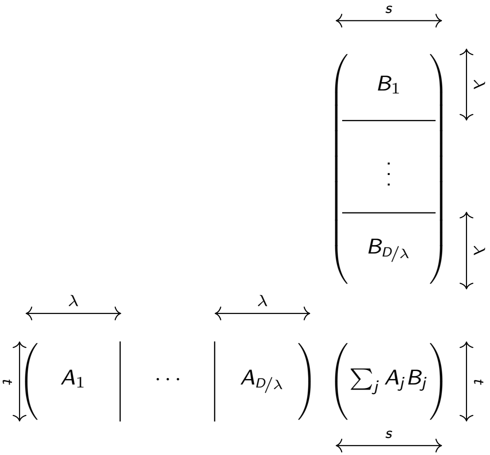
def `FFMatMulSW`:

$C = 0 \in \mathbb{F}_p^{t\times s}$

for $j = 1$ **to** $\lceil D/\lambda \rceil$ **do**

$C = (C + A_j \cdot B_j) \bmod p$

return C



Going Faster: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

Algorithm: λ-block matrix product over $\mathbb{F}_p$

Input : $A \in \mathbb{F}_p^{t \times D}$, $B \in \mathbb{F}_p^{D \times s}$, p,
$\lambda = \left\lfloor (2^{53} - p - 1)/(p-1)^2 \right\rfloor$
$\simeq 2^{53-2r}$ where $r = \texttt{bitsize}(p)$

Assumption: $a_{i,j}$, $b_{i,j} < 2^{26}$ stored as `binary64`

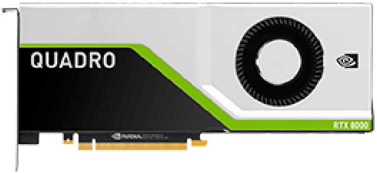
Output : $C = AB \in \mathbb{F}_p^{t \times s}$ stored as `binary64`

def `FFMatMulSW`:
  $C = 0 \in \mathbb{F}_p^{t \times s}$
  for $j = 1$ **to** $\lceil D/\lambda \rceil$ **do**
    $C = (C + A_j \cdot B_j) \bmod p$
  return C

$C + A_j \cdot B_j$: one `dgemm` instruction with a BLAS library.

- `rocBLAS`: AMD cards
- **`CUBLAS`: NVIDIA cards**

$\rightarrow$ Software implementation in `CUDA`.



Going Faster: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

Algorithm: λ-block matrix product over $\mathbb{F}_p$

Input : $A \in \mathbb{F}_p^{t \times D}$, $B \in \mathbb{F}_p^{D \times s}$, p,

$\lambda = \left\lfloor (2^{53} - p - 1)/(p-1)^2 \right\rfloor$

$\simeq 2^{53-2r}$ where $r = \texttt{bitsize}(p)$

Assumption: $a_{i,j}, b_{i,j} < 2^{26}$ stored as `binary64`

Output : $C = AB \in \mathbb{F}_p^{t \times s}$ stored as `binary64`

def `FFMatMulSW`:

$C = 0 \in \mathbb{F}_p^{t \times s}$

for $j = 1$ **to** $\lceil D/\lambda \rceil$ **do**

$C = (C + A_j \cdot B_j) \bmod p$

return C

Reduction with Fused-Multiply Add (FMA) instruction.

[Jean, Graillat 2010]

[van der Hoeven, Lecerf, Quintin 2014] (Mathemagix)

Going Faster: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet

Algorithm: λ-block matrix p

Input : $A \in \mathbb{F}_p^{t \times D}$, $B \in$

$\lambda = \left\lfloor (2^{53} - p \right.$

$\simeq 2^{53-2r}$ wher

Assumption: $a_{i,j}, b_{i,j} < 2^{26}$

Output : $C = AB \in \mathbb{F}_p^{t \times}$

def `FFMatMulSW`:

  $C = 0 \in \mathbb{F}_p^{t \times s}$

  for $j = 1$ **to** $\lceil D/\lambda \rceil$ **do**

    $C = (C + A_j \cdot B_j) \bmod p$

  return C

Why this blocksize?

$$\lambda = \left\lfloor (2^{53} - p - 1)/(p - 1)^2 \right\rfloor$$

$\simeq 2^{53-2r}$ with $r = \texttt{bitsize}(p)$

Add

n 2014]

(Mathemagix)

Delayed Modular Reduction

Maximal block size?

$$\langle u, v\rangle = (\underbrace{u_1 v_1}_{\leq (p-1)^1} + \cdots + \underbrace{u_\lambda v_\lambda}_{\leq (p-1)^1}) \bmod p + \cdots + (\underbrace{u_{D-\lambda+1} v_{D-\lambda+1}}_{\leq\ (p-1)^1} + \cdots + \underbrace{u_D v_D}_{\leq\ (p-1)^1}) \bmod p$$

Delayed Modular Reduction

Maximal block size?

$$\langle u, v\rangle = (\underbrace{\underbrace{u_1 v_1}_{\leq (p-1)^1} + \cdots + \underbrace{u_\lambda v_\lambda}_{\leq (p-1)^1}}_{\leq \lambda\cdot(p-1)^1 \leq 2^{53}}) \bmod p + \cdots + (\underbrace{\underbrace{u_{D-\lambda+1} v_{D-\lambda+1}}_{\leq (p-1)^1} + \cdots + \underbrace{u_D v_D}_{\leq (p-1)^1}}_{\leq \lambda\cdot(p-1)^1 \leq 2^{53}}) \bmod p$$

Delayed Modular Reduction

Maximal block size?

$$\langle u, v\rangle = (\underbrace{\underbrace{u_1 v_1}_{\leq (p-1)^1} + \cdots + \underbrace{u_\lambda v_\lambda}_{\leq (p-1)^1}}_{\leq\ \lambda\cdot(p-1)^1\ \leq\ 2^{53}}) \bmod p + \cdots + (\underbrace{\underbrace{u_{D-\lambda+1} v_{D-\lambda+1}}_{\leq\ (p-1)^1} + \cdots + \underbrace{u_D v_D}_{\leq\ (p-1)^1}}_{\leq\ \lambda\cdot(p-1)^1\ \leq\ 2^{53}}) \bmod p$$

Optimal lambda [Dumas, Gautier, Pernet 2002]:

$$\lambda_{\text{opt}}(p-1)^2 + (p-1) \leq 2^{53} < (\lambda_{\text{opt}}+1)(p-1)^2 + (p-1) \quad ; \quad \lambda(p) = \left\lfloor (2^{53} - p - 1)/(p-1)^2 \right\rfloor$$

Refinement:

$$\lambda(u,\ v,\ p) = \left\lfloor \frac{2^{53} - p - 1}{\max_i(u_i) * \max_j(v_j)} \right\rfloor$$

Going Further: Multi-word Computation

Two-word matrix multiplication

A_h, A_l: matrices with high/low parts resp.

$$A = 2^{r/2} \cdot A_h + A_l \quad A_h,\ A_l \in \mathbb{F}_p^{t \times D}$$
$$B = 2^{r/2} \cdot B_h + B_l \quad B_h,\ B_l \in \mathbb{F}_p^{D \times s}$$

$$C = A \cdot B = 2^r \cdot A_h \cdot B_h + 2^{r/2} \cdot (A_h \cdot B_l + A_l \cdot B_h) + A_l \cdot B_l$$

$$\lambda = \left\lfloor \frac{2^{53} - p - 1}{\max_{i,j}(a_{i,j}) * \max_{i,j}(b_{i,j})} \right\rfloor$$

$$\lambda_{ll} \le \lambda_{hl} = \lambda_{lh} \le \lambda_{hh} \simeq 2^{53-r}$$

Better than the previous 2^{53-2r}.

Going Further: Multi-word Computation

Two-word matrix multiplication

A_h, A_l: matrices with high/low parts resp.

$$A = 2^{r/2} \cdot A_h + A_l \quad A_h,\ A_l \in \mathbb{F}_p^{t \times D}$$
$$B = 2^{r/2} \cdot B_h + B_l \quad B_h,\ B_l \in \mathbb{F}_p^{D \times s}$$

$$C = A \cdot B = 2^r \cdot A_h \cdot B_h + 2^{r/2} \cdot (A_h \cdot B_l + A_l \cdot B_h) + A_l \cdot B_l$$

$$\lambda = \left\lfloor \frac{2^{53} - p - 1}{\max_{i,j}(a_{i,j}) * \max_{i,j}(b_{i,j})} \right\rfloor$$

$$\lambda_{ll} \leq \lambda_{hl} = \lambda_{lh} \leq \lambda_{hh} \simeq 2^{53-r}$$

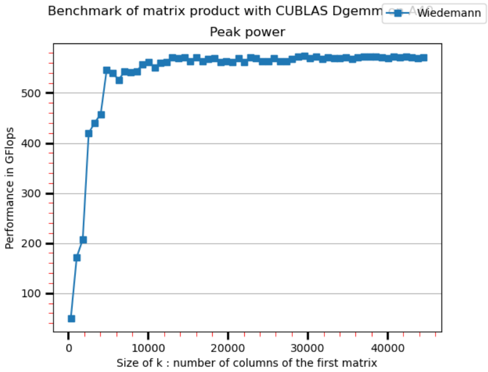
Better than the previous 2^{53-2r}. Greater prime: We can target primes $p > 2^{26}$ ✅ !

Peak Performance: Definition

$$\text{performance} = \frac{\text{N° floating-point operations}}{\text{time (s)} \cdot 10^9}$$

$$\text{performance} = 2 \cdot \frac{t \cdot D \cdot s}{\tau \cdot 10^9}$$

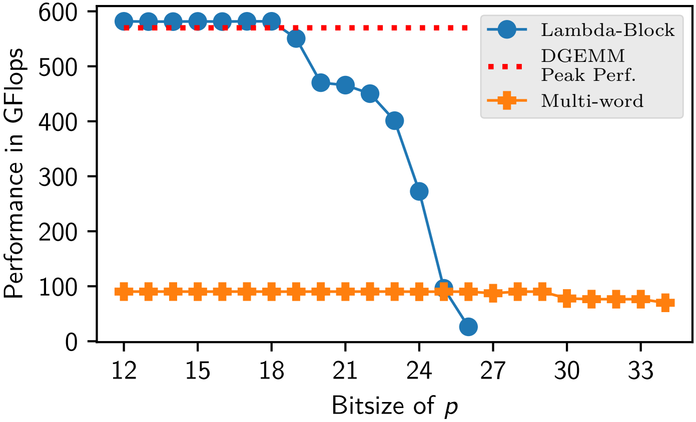
Performance in GFlops: matrix product $t \times D$ and $D \times s$



Floating-point **Rectangular** matrix multiplication on a A40 (Ampère) GPU

Peak performance at **560** GFlops on a A40

Benchmarks with Multi-word Algorithm



Comparison between single and multi-word matrix product on A40 GPU (t=15000, D=45000, s=32)
$\mathbb{F}_p$: prime field

Summary and Perspectives

Summary

- ✅ GPU Kernels for modular reduction using floating-point arithmetic in CUDA
- ✅ Modular Block-product algorithm implemented using CUBLAS
- ✅ Multi-word algorithm with floating-point arithmetic for primes larger than 26-bit

Perspectives

- Theoretical Peak Performance (GFlops) on A40 (1:64 ratio)

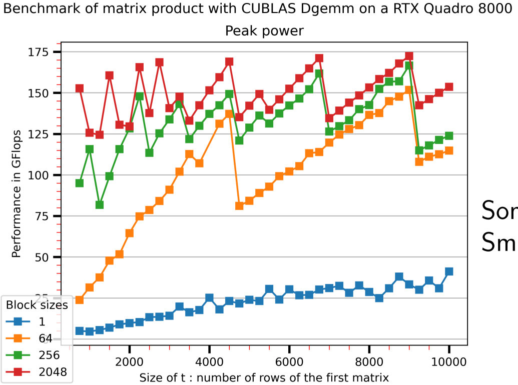
Precisions	binary32 (GFlops)	binary64 (GFlops)	ratio (b32/b64)
A40	37420	585	64

$\Rightarrow$ Split even more to use binary32 in multi-word algorithm

- Integrate in MSOLVE `https://github.com/algebraic-solving/msolve`

Thanks for your attention!

How Much Time Does A Block-Product Take?



$$\text{performance} = 2 \cdot \frac{t \cdot \lambda \cdot s}{\tau \cdot 10^9}$$

Some performance drops as D increases

Small blocksize: huge performance impact