Multiword matrix decomposition with floating-point representation for modular matrix product

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

LIP6 — Sorbonne Université – Université Paris Cité – CNRS

ODELIX Thematic Fall Program — 24 novembre 2025

Context and Objectives

Multimodular Linear Algebra

Problems over $\mathbb{Z}$ or $\mathbb{Q}$: GCD, Linear System Solving, ...
Matrix product: basic block of linear algebra
Exact computation: Huge intermediate rational coefficients!
Solution: Computations over several **prime fields** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} = \mathbb{F}_p$

Context and Objectives

Multimodular Linear Algebra

Problems over $\mathbb{Z}$ or $\mathbb{Q}$: GCD, Linear System Solving, ...
Matrix product: basic block of linear algebra
Exact computation: Huge intermediate rational coefficients!
Solution: Computations over several **prime fields** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} = \mathbb{F}_p$

Going Faster

Leverage **multi-core CPU and GPU**

Matrix product efficiently done with **BLAS**:
floating-point arithmetic only

Context and Objectives

Multimodular Linear Algebra

Problems over $\mathbb{Z}$ or $\mathbb{Q}$: GCD, Linear System Solving, ...
Matrix product: basic block of linear algebra
Exact computation: Huge intermediate rational coefficients!
Solution: Computations over several **prime fields** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} = \mathbb{F}_p$

Going Faster

Leverage **multi-core CPU and GPU**

Matrix product efficiently done with **BLAS**:
floating-point arithmetic only

Going Further

double: $2^{53} \rightarrow p < 2^{26}$ ☹

Goal: Lift the prime limit of 26 bits while preserving efficiency

$\rightarrow$ **Multiword** matrix product over $\mathbb{F}_p$

Summary

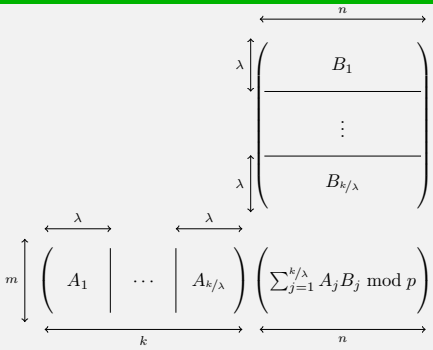
State-of-the-Art: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, p, λ
Output : $C = AB \in \mathbb{F}_p^{m \times n}$

$C = 0 \in \mathbb{F}_p^{m \times n}$
for $j = 1$ **to** $\lceil k/\lambda \rceil$ **do**
    $C = (C + A_j B_j) \bmod p$



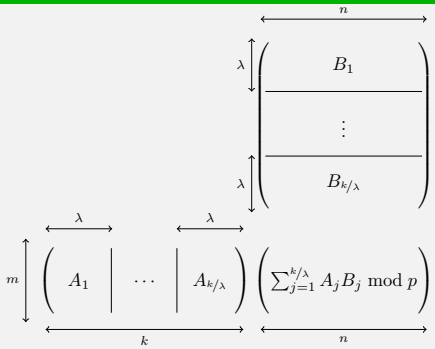
State-of-the-Art: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, p, λ
Output : $C = AB \in \mathbb{F}_p^{m \times n}$

$C = 0 \in \mathbb{F}_p^{m \times n}$
for $j = 1$ **to** $\lceil k/\lambda \rceil$ **do**
$C = (C + A_j B_j) \bmod p$



1. Separates matrix product from modular reductions: `dgemm` routine from **BLAS** for $C + A_j B_j$

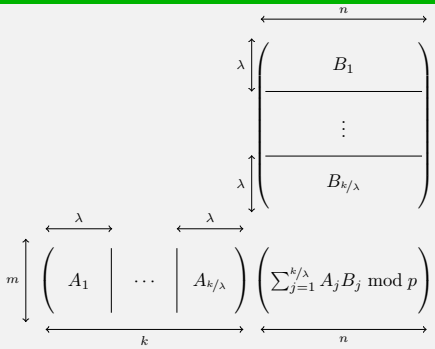
State-of-the-Art: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, p, λ
Output : $C = AB \in \mathbb{F}_p^{m \times n}$

$C = 0 \in \mathbb{F}_p^{m \times n}$
for $j = 1$ **to** $\lceil k/\lambda \rceil$ **do**
    $C = (C + A_j B_j) \bmod p$



1. Separates matrix product from modular reductions: `dgemm` routine from **BLAS** for $C + A_j B_j$
2. Minimizes the number of reductions to $\lceil \frac{k}{\lambda} \rceil mn$, λ large $\rightarrow$ same perf as `dgemm`

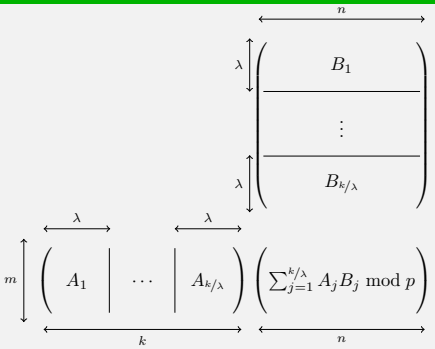
State-of-the-Art: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, p, λ
Output : $C = AB \in \mathbb{F}_p^{m\times n}$

$C = 0 \in \mathbb{F}_p^{m\times n}$
for $j = 1$ **to** $\lceil k/\lambda \rceil$ **do**
$C = (C + A_j B_j) \bmod p$



1. Separates matrix product from modular reductions: `dgemm` routine from **BLAS** for $C + A_j B_j$
2. Minimizes the number of reductions to $\lceil \frac{k}{\lambda} \rceil mn$, λ large $\rightarrow$ same perf as `dgemm`
3. Keeps result exact when λ is small enough $\rightarrow \lambda_{\mathbf{MAX}} = \left\lfloor \frac{2^t - p + 1}{(p-1)^2} \right\rfloor$ (t: mantissa's bitsize)

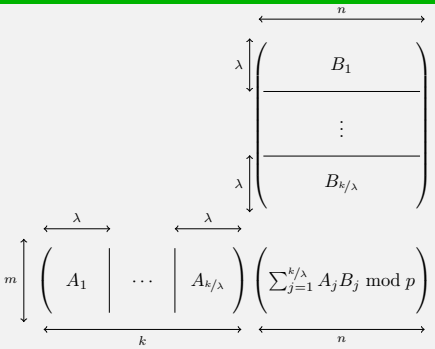
State-of-the-Art: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

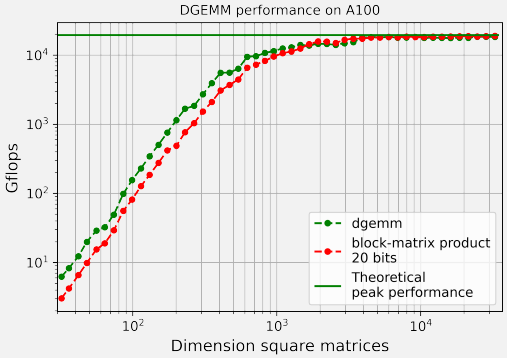
Input : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, p, λ
Output : $C = AB \in \mathbb{F}_p^{m\times n}$

$C = 0 \in \mathbb{F}_p^{m\times n}$
for $j = 1$ **to** $\lceil k/\lambda \rceil$ **do**
$C = (C + A_j B_j) \bmod p$

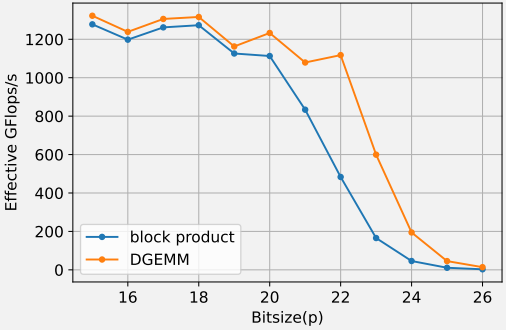


1. Separates matrix product from modular reductions: dgemm routine from **BLAS** for $C + A_j B_j$
2. Minimizes the number of reductions to $\lceil \frac{k}{\lambda} \rceil mn$, λ large $\rightarrow$ same perf as dgemm
3. Keeps result exact when λ is small enough $\rightarrow \lambda_{\mathbf{MAX}} = \left\lfloor \frac{2^t - p + 1}{(p-1)^2} \right\rfloor$ (t: mantissa's bitsize)
4. For bitsizes larger than 26 bits $\lambda_{\mathbf{MAX}} = 0$. RNS [Doliskani, Giorgi, Lebreton, Schost 2018]

Modular reduction is costly



Variation of performance per matrix dimensions of the block-matrix product on GPU



Comparison of performance of Block-matrix product and DGEMM per prime bit size on CPU

Floating-point arithmetic

Standard model of IEEE 754

In absence of underflow or overflow,
$\mathrm{op} \in \{+, -, \times, /\}$, fl a rounding mode:

$$\mathrm{fl}(a \,\mathrm{op}\, b) = (a \,\mathrm{op}\, b)(1+\eta), \quad |\eta| \leq \varepsilon/(1+\varepsilon)$$

where $\varepsilon = 2^{-t}$: unit roundoff
This is a **uniform bound** on the relative fp **error**, **independent** of the rounding mode

Representable floating-point integers

Representation: only integer coefficients using floating-point arithmetic
Implies $p < 2^t$ for every coefficient to be exactly represented Otherwise, integer x is exact iff:

1. $2^{-t} < x < 2^t$
2. $x > 2^t, x \equiv 0 \bmod 2^{e-52}$
3. $x < 2^t, x \equiv 0 \bmod 2^{e+52}$

Assumption: All operation results must lie in $\mathbb{F} = \{0, 1, \ldots, 2^t\}$

Modular reduction using floating-point arithmetic

Input : $x \in \mathbb{F} < 2^t$, a prime number
$p \in \mathbb{F} < 2^{t-1}$, and $q = \mathrm{fl}(1/p)$
Output: $d \in \mathbb{F}_p$ such that $d = x \bmod p$
$b = xq$
$c = \lfloor b \rfloor$
$d = x - cp$
if $d \geq p$ **then**
  $d = d - p$
if $d < 0$ **then**
  $d = d + p$

Algorithm 1: Floating-point reduction
[van der Hoeven, Quintin, Lecerf, 2017]

Floating-point reduction without FMA [vdH,Q,L]

floating-point reduction without fma is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $p < 2^{(t-1)/2}$ and $x \leq 2^{(t-1)/2} p$

- Inverse precomputed for an array
- only 5 ops and 2 comparisons at max.

Modular reduction using floating-point arithmetic

Input : $x \in \mathbb{F} < 2^t$, a prime number
$p \in \mathbb{F} < 2^{t-1}$, and $q = \mathrm{fl}(1/p)$
Output: $d \in \mathbb{F}_p$ such that $d = x \bmod p$
$b = xq$
$c = \lfloor b \rfloor$
$d = x - cp$
if $d \geq p$ **then**
  $d = d - p$
if $d < 0$ **then**
  $d = d + p$

Algorithm 1: Floating-point reduction
[van der Hoeven, Quintin, Lecerf, 2017]

Floating-point reduction without FMA [vdH,Q,L]

floating-point reduction without fma is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $p < 2^{(t-1)/2}$ and $x \leq 2^{(t-1)/2} p$

Floating-point reduction without FMA

floating-point reduction is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $4 \leq p < 2^{t-1}$ and $x \leq 2^{t-1} - p$

Modular reduction using floating-point arithmetic

Input : $x \in \mathbb{F} < 2^t$, a prime number
$p \in \mathbb{F} < 2^{t-1}$, and $q = \mathrm{fl}(1/p)$
Output: $d \in \mathbb{F}_p$ such that $d = x \bmod p$
$b = xq$
$c = \lfloor b \rfloor$
$d = \mathtt{fma}(-c, p, x)$ // $x - cp$
if $d \geq p$ **then**
    $d = d - p$
if $d < 0$ **then**
    $d = d + p$

Algorithm 1: Floating-point reduction
[van der Hoeven, Quintin, Lecerf, 2017]

Floating-point reduction correctness

floating-point reduction is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $4 \leq p < 2^{t-1}$ and $x \leq 2^{t-2}p$

Modular reduction using floating-point arithmetic

Input : $x \in \mathbb{F} < 2^t$, a prime number
$p \in \mathbb{F} < 2^{t-1}$, and $q = \mathrm{fl}(1/p)$
Output: $d \in \mathbb{F}_p$ such that $d = x \bmod p$
$b = xq$
$c = \lfloor b \rfloor$
$d = \mathtt{fma}(-c, p, x)$ // $x - cp$
if $d \geq p$ **then**
    $d = d - p$
if $d < 0$ **then**
    $d = d + p$

Algorithm 1: Floating-point reduction
[van der Hoeven, Quintin, Lecerf, 2017]

Floating-point reduction correctness

floating-point reduction is correct for any integer input $x \in \mathbb{F}$ and a modulus p such that $4 \leq p < 2^{t-1}$ and $x \leq 2^{t-2}p$

- With FMA, all integers $0 \leq x < 2^t$ can be reduced $\bmod p \geq 4$
- No need for integer techniques: Barrett or Montgomery

Modular multiplication using floating-point arithmetic

Input : $x, y \in \mathbb{F}$ s.t. $xy \leq 2^{t-2}p$,
$p \in \mathbb{F} \leq 2^{t-1}$, $q = \mathrm{fl}(1/p)$
Output: $e \in \mathbb{F}$ such that $e = xy \bmod p$
$h = \mathrm{fl}(xy)$
$l = \mathtt{fma}(x, y, -h)$// $xy - h$, fp error
$b = \mathrm{fl}(hq)$
$c = \lfloor b \rfloor$
$d = \mathtt{fma}(-c, p, h)$// $h - cp$
$e = \mathrm{fl}(d + l)$// Error-free transformation
if $e \geq p$ **then**
    $e = e - p$
if $e < 0$ **then**
    $e = e + p$

Algorithm 2: MODMUL [van der Hoeven, Quintin, Lecerf](Function 3.6)

MODMUL correctness

MODMUL is correct for integer input x and y in $\mathbb{F}$ such that their product satisfies

$$xy \leq \frac{2^{t-1}}{3}p$$

and for input $p \leq 2^{t-1}$, for all $t \geq 3$

Multiword Algorithm

$A \in \mathbb{F}_p^{m\times k}, B \in \mathbb{F}_p^{k\times n}$, targets p prime with $f := \texttt{bitsize}\, p, \quad t/2 \le f \le t-1$ bits ($\texttt{double}: 26 \le f \le 52$).

Multiword Matrix Algorithm

New Algorithm à la Ozaki scheme, in 3 steps:

1. Split each input matrices into several matrices (matrices of words) with smaller coefficients (at most $t/2$ bits)
2. Compute the outer product of the vectors of matrices of words
3. Combine the matrix products into the resulting product $C = AB$

We have changed our multiword representation:

1. $A = \sum_{i=0}^{u-1} A_i (2^{\frac{t}{u}})^i$ [Berthomieu, Graillat, L., Mary, 2023] (hardly generalizable)
2. $A = \sum_{i=0}^{u-1} A_i \left(\lceil p^{\frac{1}{u}} \rceil \right)^i$ [Berthomieu, Graillat, L., Mary, 2025 (preprint)]

Multiword Decomposition

$A \in \mathbb{F}_p^{m \times k}, B \in \mathbb{F}_p^{k \times n}$, p prime with $f = \texttt{bitsize}\, p > 26$ bits.

(u, v)-Multiword matrix decomposition

$1 \leq u, v \leq 3$

$\alpha := \left\lceil p^{1/u} \right\rceil, \quad \beta := \left\lceil p^{1/v} \right\rceil$

$$A = \sum_{i=0}^{u-1} \alpha^i A_i$$

Smaller **Coefficients** of:

- A_i **bounded by** α,
- B_j **bounded by** β.

Multiword Decomposition

$A \in \mathbb{F}_p^{m \times k}, B \in \mathbb{F}_p^{k \times n}$, p prime with $f = \texttt{bitsize}\, p > 26$ bits.

(u, v)-Multiword matrix decomposition

$1 \leq u, v \leq 3$

$\alpha := \lceil p^{1/u} \rceil, \quad \beta := \lceil p^{1/v} \rceil$

$$A = \sum_{i=0}^{u-1} \alpha^i A_i$$

Smaller **Coefficients** of:

- A_i **bounded by** α,
- B_j **bounded by** β.

In $\mathbb{F}$ floor-div is exact

- $(a, b) \in \mathbb{N}^2, 0 \leq a < 2^t, 1 \leq b \in \mathbb{N}$,

$$\lfloor \mathrm{fl}(a/b) \rfloor = \lfloor a/b \rfloor$$

- Decomposition is exactly an α-basis integer decomposition without floating-point approximations

Reconstructions

Objective: Compute $C = \sum_{i,j}^{u-1,v-1} \alpha^i \beta^j A_i B_j$

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

$C = 0$
for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $\gamma = \alpha^i \beta^j \bmod p$
    $T = A_i B_j$ // λ-block matmul
    $T = \gamma T$ // MODMUL
    $C = C + T$

Problem: T additional workspace
memory management is slow on GPUs

Reconstructions

Objective: Compute $C = \sum_{i,j}^{u-1,v-1} \alpha^i \beta^j A_i B_j$

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

$C = 0$
for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $\gamma = \alpha^i \beta^j \bmod p$
    $T = A_i B_j$ // λ-block matmul
    $T = \gamma T$ // MODMUL
    $C = C + T$

Problem: T additional workspace
memory management is slow on GPUs

Finite Field $\mathbb{F}_p$ reconstruction

$C = 0$
for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $\gamma = \alpha^i \beta^j \bmod p$ // MODMUL
    $\delta = \gamma^{-1} \bmod p$ // XGCD
    $C = \delta C \bmod p$ // MODMUL
    $C = C + A_i B_j$ // λ-block matmul
    $C = \gamma C \bmod p$ // MODMUL

Multiword product: Prime limit

Theorem: Prime limit

The (u, v)-multiword product is correct for primes with at most $t\frac{uv}{u+v}$ bits (t: mantissa bitsize).

Multiword product: Prime limit

Theorem: Prime limit

The (u, v)-multiword product is correct for primes with at most $t\frac{uv}{u+v}$ bits (t: mantissa bitsize).
The optimal delay parameter is $\lambda = \lfloor (2^t - p + 1)/(\alpha\beta) \rfloor$

Multiword product: Prime limit

Theorem: Prime limit

The (u, v)-multiword product is correct for primes with at most $t\frac{uv}{u+v}$ bits (t: mantissa bitsize).
The optimal delay parameter is $\lambda = \lfloor (2^t - p + 1)/(\alpha\beta) \rfloor$

Limited by modular reduction ($t - 1$ bits) anyway!

(u, v)	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(2,2)$	$(2,3)$
uv: # blockproducts	1	2	3	4	4	6
Limit on $\texttt{bitsize}\,(p)$	$t/2$	$2t/3$	$3t/4$	$4t/5$	$t-1$	$t-1$
Limit, $t = 53$	26	35	39	42	52	52

Multiword product: Prime limit

Theorem: Prime limit

The (u, v)-multiword product is correct for primes with at most $t\frac{uv}{u+v}$ bits (t: mantissa bitsize).
The optimal delay parameter is $\lambda = \lfloor (2^t - p + 1)/(\alpha\beta) \rfloor$

Limited by modular reduction ($t - 1$ bits) anyway!

(u, v)	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(2,2)$	$(2,3)$
uv: # blockproducts	1	2	3	4	4	6
Limit on $\texttt{bitsize}(p)$	$t/2$	$2t/3$	$3t/4$	$4t/5$	$t-1$	$t-1$
Limit, $t = 53$	26	35	39	42	52	52

$(2, 2)$: works with almost any prime in $\mathbb{F}$
$(2, 3)$: Greater blocksize than $(2, 2)$ for the largest primes.

Experimental setting

Performance metric

Input matrices of sizes $m \times k$ and $k \times n$: Effective GFlops/s: $2mnk/\text{time}$
Measure number of **Field operations** and not floating-point instructions per second.

Experimental setting

Performance metric

Input matrices of sizes $m \times k$ and $k \times n$: Effective GFlops/s: $2mnk/\text{time}$
Measure number of **Field operations** and not floating-point instructions per second.

Decomposition Time

- Square matrices: $m = k = n = 10016$
  both input matrices
- Unbalanced matrices: $m = 11000$, $k = 33000$, $n = 32$
  right input matrix only as an iterative context is assumed: decomposition is reused anyway

Experimental setting

Performance metric

Input matrices of sizes $m \times k$ and $k \times n$: Effective GFlops/s: $2mnk/\text{time}$
Measure number of **Field operations** and not floating-point instructions per second.

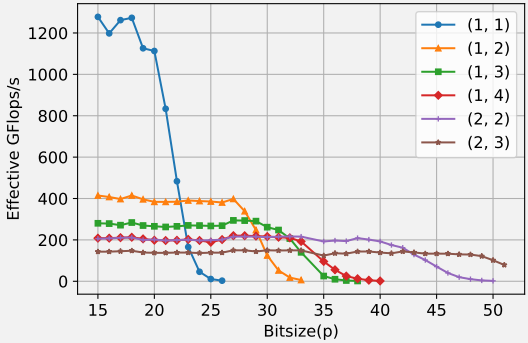
Decomposition Time

- Square matrices: $m = k = n = 10016$
  both input matrices
- Unbalanced matrices: $m = 11000$, $k = 33000$, $n = 32$
  right input matrix only as an iterative context is assumed: decomposition is reused anyway

- GPU: CUDA & CUBLAS, NVIDIA A100, 3456 fp64 cores
- CPU: Fortran90 & MKL, Intel Xeon Gold 6248, 40 threads

Square matrices: $(m = k = n = 10016)$

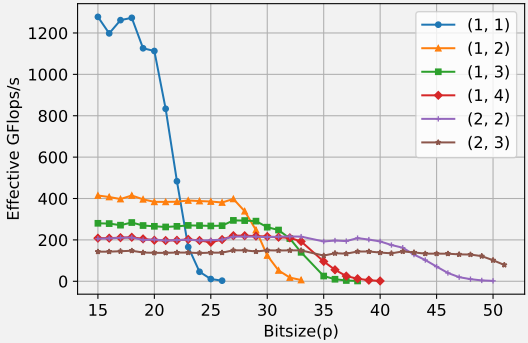
- Constant Performance for bitsizes lower than 16 bits



Our implementation on Intel Xeon Gold 6248 CPU

Square matrices: $(m = k = n = 10016)$

- Constant Performance for bitsizes lower than 16 bits
- p increases $\rightarrow$ more splitting
  $\rightarrow$ more reductions, performance decreases



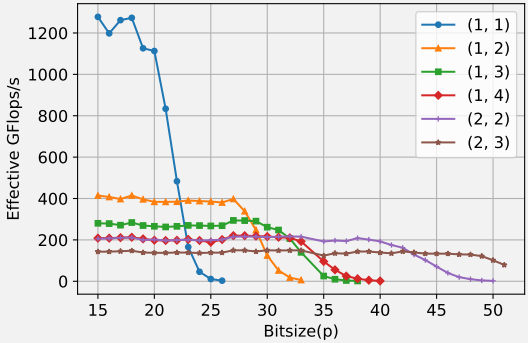
Our implementation on Intel Xeon Gold 6248 CPU

Square matrices: $(m = k = n = 10016)$

- Constant Performance for bitsizes lower than 16 bits
- p increases $\rightarrow$ more splitting
  $\rightarrow$ more reductions, performance decreases
- Best variant for:

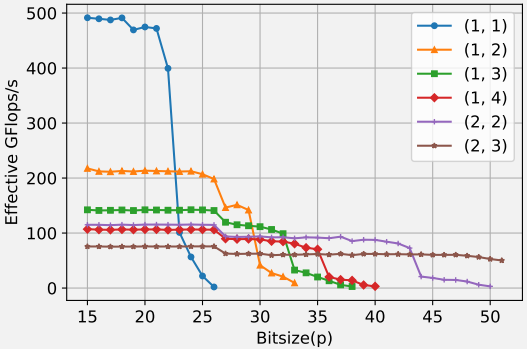
$(1,2)$	$[23,29]$ bits,
$(1,3)$	$[30,33]$ bits,
$(2,2)$	$[34,43]$ bits,
$(2,3)$	$[44,52]$ bits.

- $(1,2)$ better than $(1,1)$ product for $[23,26]$ bits!

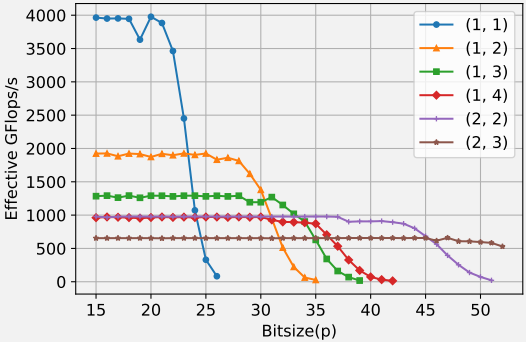


Our implementation on Intel Xeon Gold 6248 CPU

Unbalanced matrices: (m=11000, k=33000, n=32)

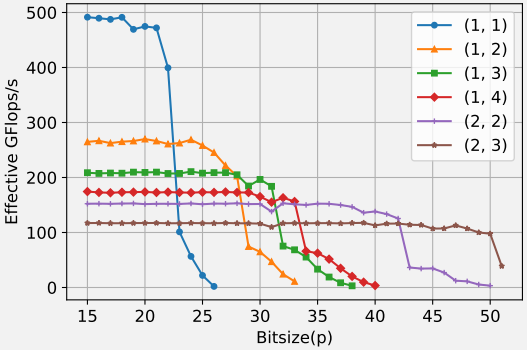


on Intel Xeon Gold 6248 CPU

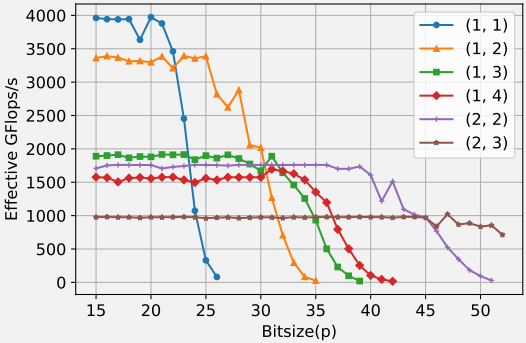


on NVIDIA A100 GPU

Unbalanced matrices: (m=11000, k=33000, n=32)



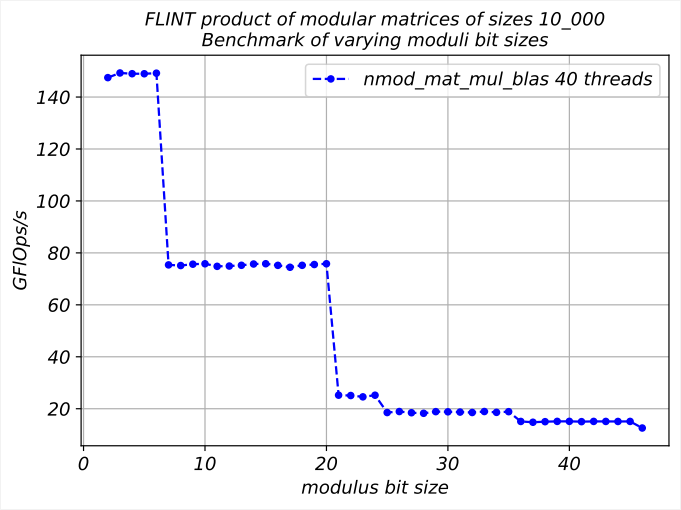
on Intel Xeon Gold 6248 CPU



on NVIDIA A100 GPU

With concatenation

FLINT


FLINT product of modular matrices of sizes 10_000
Benchmark of varying moduli bit sizes
nmod_mat_mul_blas 40 threads
GFlOps/s
140
120
100
80
60
40
20
0
10
20
30
40
modulus bit size

Residue Number System

Lifting success condition

Matrix product over $\mathbb{F}_p$ can be computed using CRT if: $k(p-1)^2 \leq N$, N being the RNS modulus

RNS decomposition

Let $\forall\ 1 \leq i \leq s$, m_i be pairwise coprime moduli with $N = \prod_{i=1}^{s} m_i > k(p-1)^2$.
A_i contains coefficients of A modulo m_i. Same for B_i and B.

- A_iB_i are computed modulo $\forall i$, m_i.
  s subproducts.
- $C = AB \bmod p$ is reconstructed using the CRT from the $C_i = A_iB_i \bmod m_i$.

s must be large enough so that $m_i = \Theta(N/s) < 2^{26}$

Residue Number System

Number moduli RNS matrix product

Assuming enough pairwise coprime moduli exist, s number of moduli required to compute a matrix product $C = AB$, $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, satisfies:

$$s = \left\lceil \frac{2(d + 2f)}{t - \ell} \right\rceil$$

- $d = \lceil \log(k) \rceil$
- $f = \log(p - 1)$
- t: underlying type precision
- $\ell = \lfloor \log(\lambda) \rfloor$, λ: minimal delay parameter

Residue Number System

Number moduli RNS matrix product

Assuming enough pairwise coprime moduli exist, s number of moduli required to compute a matrix product $C = AB$, $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, satisfies:

$$s = \left\lceil \frac{2(d + 2f)}{t - \ell} \right\rceil$$

- $d = \lceil \log(k) \rceil$
- $f = \log(p - 1)$
- t: underlying type precision
- $\ell = \lfloor \log(\lambda) \rfloor$, λ: minimal delay parameter

RNS:

$$s > \frac{2d + 4f}{t - l}$$

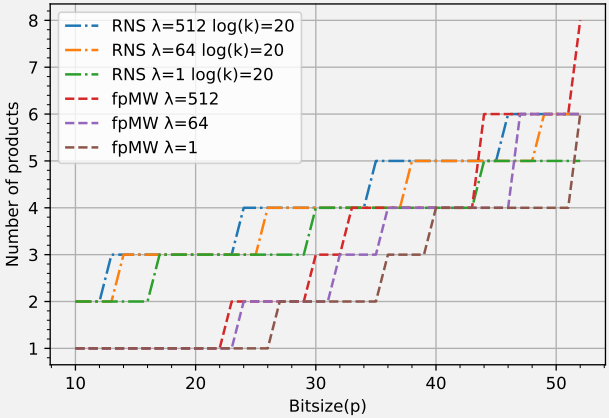
fpMW:

$$uv > \frac{(u + v)f}{t - l}$$

fpMW needs fewer products when:

1. at most $(2, 2)$-variant selected
2. $(2, 3)$-variant selected and $p < k^2$

Comparison fp Multiword versus RNS



- For bitsizes under 40 with same delay parameter, our approach requires less products
- the number of products of fpMW does not depend on k.

Perspectives

Code is online! 

- SOON Application to a Block-Wiedemann algorithm: Asynchronous kernel
- SOON Lower/Mixed-precision variants: GPU Tensor Cores
  FP16 accumulated into FP32, INT8 into INT32

Thanks for your attention!