Multiword Matrix Multiplication over Large Prime Fields on GPUs

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

LIP6 — Sorbonne Université – CNRS

PEQUAN meeting, 24 June 2023, Paris

Context and Objectives

Computer 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** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} \simeq \mathbb{F}_p$

Context and Objectives

Computer Algebra

Problems over $\mathbb{Z}$ or $\mathbb{Q}$: GCD, Linear System Solving, ...

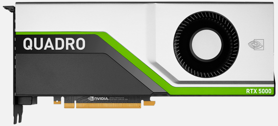
Matrix product: basic block of linear algebra

Exact computation: Huge intermediate rational coefficients!

Solution: Computations over several **prime fields** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} \simeq \mathbb{F}_p$

Going Faster

Leverage **multi-core CPU or GPU**



Matrix product efficiently done with

$\rightarrow$ **BLAS**: **floating-point** arithmetic only

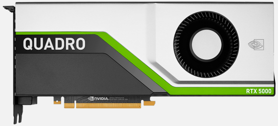
Context and Objectives

Computer 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** $\mathbb{Z}/p\mathbb{Z} = \{0, \ldots, p-1\} \simeq \mathbb{F}_p$

Going Faster

Leverage **multi-core CPU or GPU**



Matrix product efficiently done with
$\rightarrow$ **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 $\mathbb{F}_p$

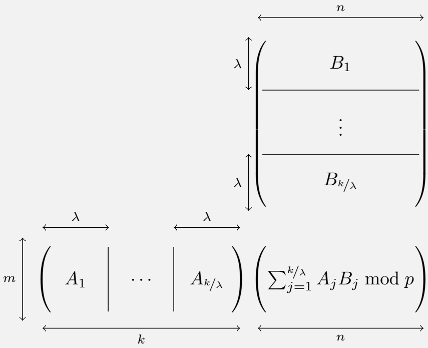
Going Faster: Matrix Multiplication over Prime Fields

[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, 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**
    $C = (C + A_j B_j) \bmod p$



Going Faster: Matrix Multiplication over Prime Fields

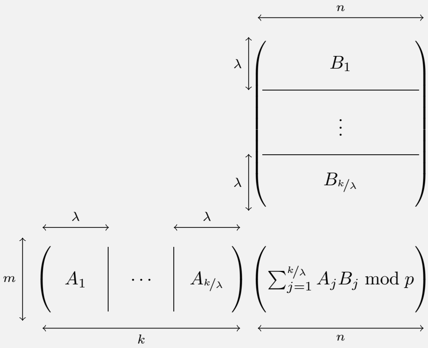
[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$, 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**
    $C = (C + A_j B_j) \bmod p$

on CPU: LINBOX
on GPU: our implementation



Going Faster: Matrix Multiplication over Prime Fields

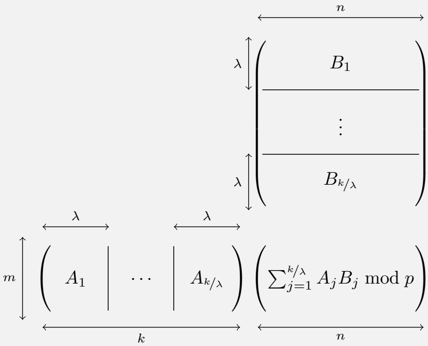
[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, 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**
$\quad C = (C + A_j B_j) \bmod p$

on CPU: LINBOX
on GPU: our implementation



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

Going Faster: Matrix Multiplication over Prime Fields

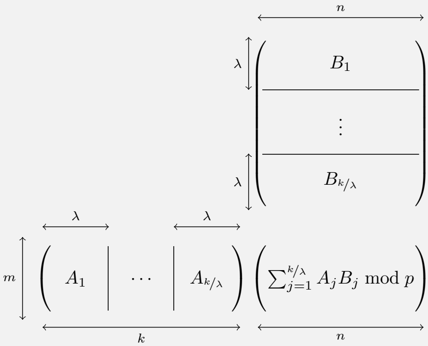
[Dumas, Gautier, Pernet 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, 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**
  $C = (C + A_j B_j) \bmod p$

on CPU: LINBOX
on GPU: our implementation



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

Going Faster: Matrix Multiplication over Prime Fields

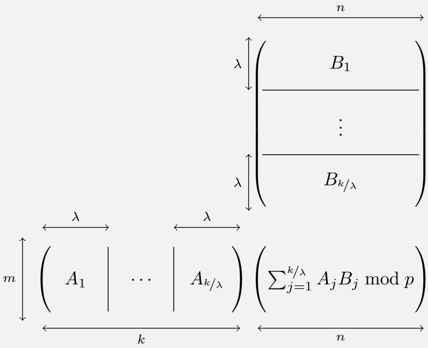
[DUMAS, GAUTIER, PERNET 2002]

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

Input : $A \in \mathbb{F}_p^{m \times k}$, $B \in \mathbb{F}_p^{k \times n}$, 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**
    $C = (C + A_j B_j) \bmod p$

on CPU: LINBOX
on GPU: our implementation



1. Separates matrix product from modular reductions: `dgemm` routine from **BLAS** for $C + A_j B_j$
2. Minimize the number of reductions $\lceil \frac{k}{\lambda} \rceil mn$ reductions, 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)

Going Further: Multiword Computation

(u, v)-Multiword matrix decomposition

$$A = \sum_{i=0}^{u-1} \alpha^i A_i \quad \alpha := \left\lceil p^{1/u} \right\rceil$$

$$B = \sum_{j=0}^{v-1} \beta^j B_j \quad \beta := \left\lceil p^{1/v} \right\rceil$$

Smaller **Coefficients** of:

- A_i **bounded by** $\alpha + 1$,
- B_j **bounded by** $\beta + 1$.

Going Further: Multiword Computation

(u, v)-Multiword matrix decomposition

$$A = \sum_{i=0}^{u-1} \alpha^i A_i \quad \alpha := \left\lceil p^{1/u} \right\rceil$$

$$B = \sum_{j=0}^{v-1} \beta^j B_j \quad \beta := \left\lceil p^{1/v} \right\rceil$$

Smaller **Coefficients** of:

- A_i **bounded by** $\alpha + 1$,
- B_j **bounded by** $\beta + 1$.

(u, v)-Multiword matrix multiplication

uv block-products of size $m \times k \times n$

for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $C = (C + \alpha^i \beta^j (A_i B_j \bmod p)) \bmod p$

Going Further: Multiword Computation

(u, v)-Multiword matrix decomposition

$$A = \sum_{i=0}^{u-1} \alpha^i A_i \quad \alpha := \left\lceil p^{1/u} \right\rceil$$

$$B = \sum_{j=0}^{v-1} \beta^j B_j \quad \beta := \left\lceil p^{1/v} \right\rceil$$

Smaller **Coefficients** of:

- A_i **bounded by** $\alpha + 1$,
- B_j **bounded by** $\beta + 1$.

(u, v)-Multiword matrix multiplication

uv block-products of size $m \times k \times n$

for $i = 0$ **to** $u - 1$ **do**
  for $j = 0$ **to** $v - 1$ **do**
    $C = (C + \alpha^i \beta^j (A_i B_j \bmod p)) \bmod p$

Product with concatenation

u block-products of size $m \times k \times vn$

for $i = 0$ **to** $u - 1$ **do**
  $[T_0, \ldots, T_v] = A_i [B_0, \ldots, B_v] \bmod p$
  for $j = 0$ **to** $v - 1$ **do**
    $C = (C + \alpha^i \beta^j T_j) \bmod p$

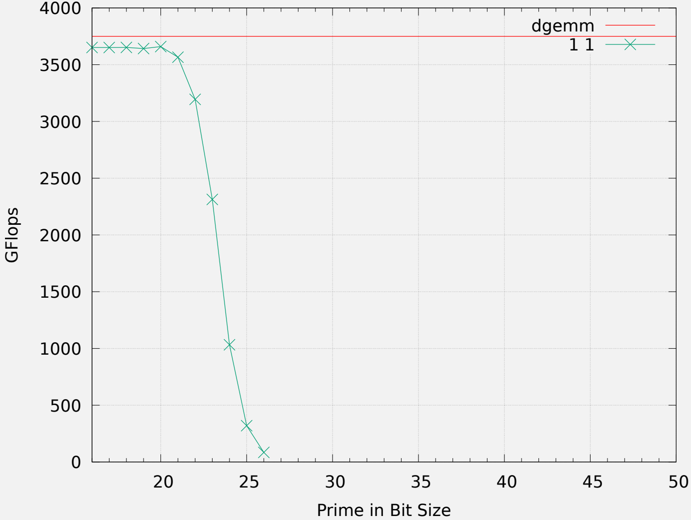
Multiword product: Prime limit

Theorem: Prime limit

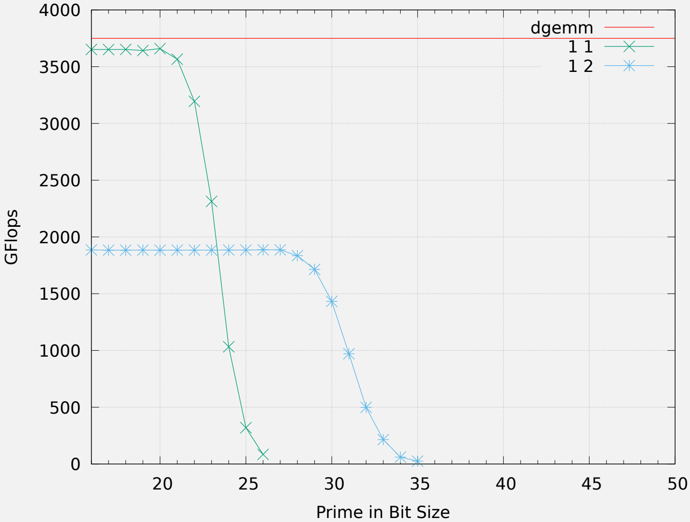
The (u, v)-multiword product is correct for primes with at most $t\frac{uv}{u+v}$ bits (t: mantissa bitsize).

(u, v)	$(1, 1)$	$(1, 2)$	$(1, 3)$	$(1, 4)$	$(2, 2)$	$(2, 3)$
uv: # blockproducts	1	2	3	4	4	6
Limit on $\log_2 p$	$t/2$	$2t/3$	$3t/4$	$4t/5$	t	$6t/5$
Limit, $t = 53$	26	35	39	42	53	63

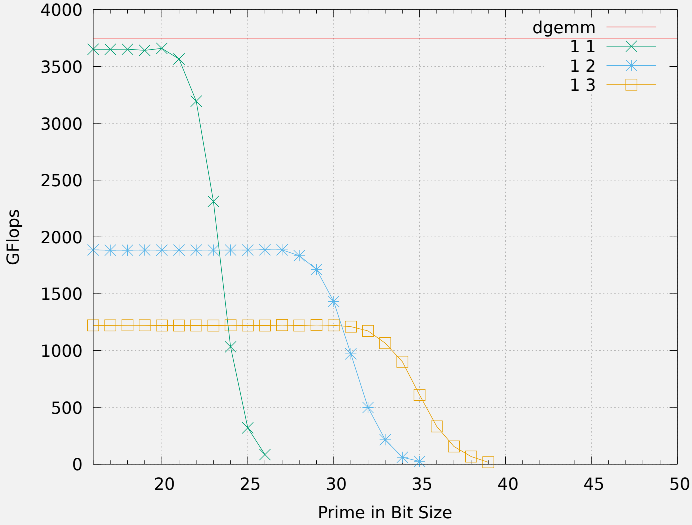
$(2, 2)$: works with any prime representable on a floating-point type
$(2, 3)$: Greater blocksize than $(2, 2)$ for the largest prime.



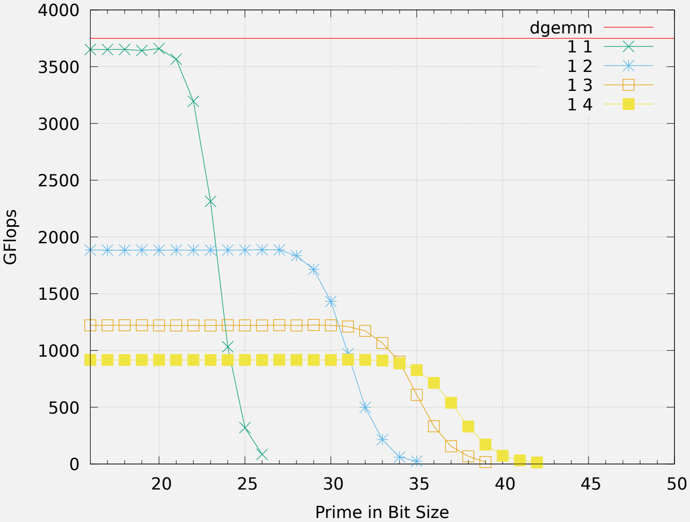
Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.



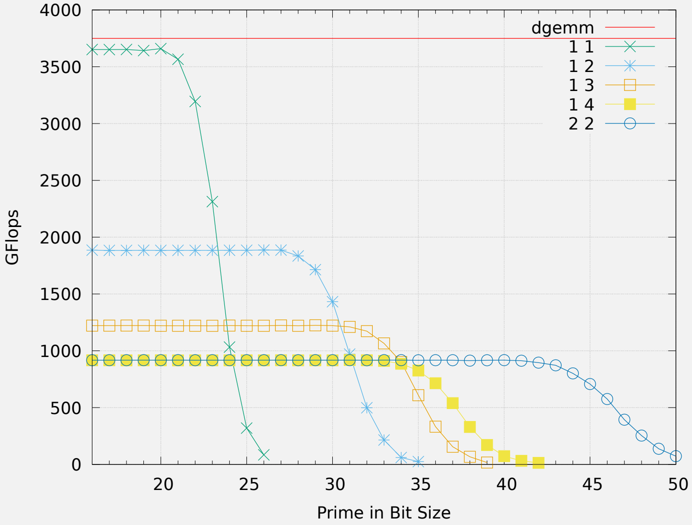
Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.



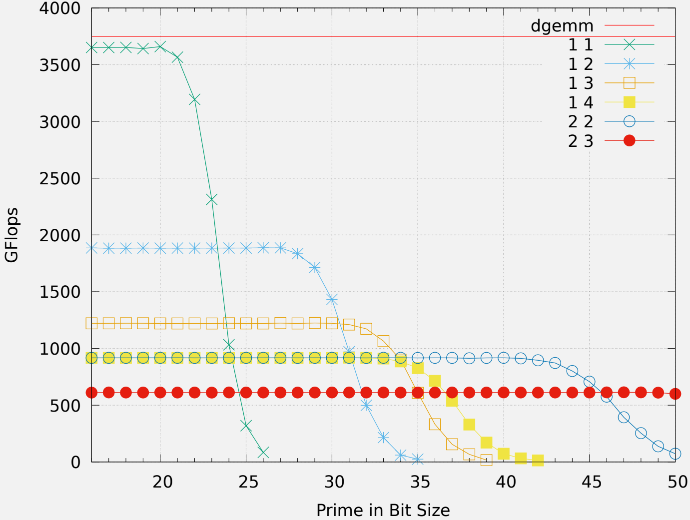
Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.



Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.



Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.



Performance of single and multi-word matrix product on NVIDIA A100 GPU (m=11000, k=33000, n=32)
A100 Theoretical peak performance: **9746** GFlops.

Take home messages

- ✅ Modular Block-product algorithm implemented using CUBLAS (NVIDIA GPUs)
- ✅ Multiword algorithms with floating-point arithmetic for primes up to 50-bit

Future work

- 🔜 Lower/Mixed-precision variants:
  GPU Tensor Cores FP16 accumulated into FP32, INT8 into INT32

Thanks for your attention!