Modular matrix multiplication on GPU for polynomial system solving

Jérémy Berthomieu, Stef Graillat, Dimitri Lesnoff, Theo Mary

Modular matrix multiplication on GPU for polynomial system solving

Jérémy Berthomieu, Stef Graillat, Dimitri Lesnoff, Theo Mary
Sorbonne Université, CNRS, LIP6, Paris, France
jeremy.berthomieu@lip6.fr, stef.graillat@lip6.fr
dimitri.lesnoff@lip6.fr, theo.mary@lip6.fr

June 5, 2023

Abstract

The bottleneck of the sparse-FGLM algorithm for Gröbner bases change of order is an iterative matrix – tall and skinny matrix product over a finite prime field. Our contribution is twofold. First, we port existing CPU-only algorithms for matrix products over prime fields to GPU architectures, and carry out a performance analysis of our implementation that shows that we can nearly achieve the maximum theoretical throughput of the hardware. Second, existing CPU-only algorithms could not handle primes with more than 26 bits, other than the GMP-based implementation in FLINT; we overcome this limitation by proposing an efficient multiword matrix product algorithm that can deal with primes with at most 35 bits; we benchmarked it on GPU.

1 Introduction

Motivation. Many problems from scientific domains, such as biology, chemistry, quantum mechanics, robotics, and computing sciences, including coding theory, computer vision and cryptography, to cite a few, can be modeled with polynomial systems. Yet, polynomial system solving is NP-hard, even when the ground field is finite.

To circumvent reliability issues from numerical algorithms, such as the number of computed solutions or the quality of the approximations of the solutions, we focus on solving exactly the 0-dimensional polynomial input system. This comes down to providing a complete description of the solution set through a lexicographic Gröbner basis. This Gröbner basis is computed in two steps: first, a Gröbner basis for a total degree order is computed using Faugère's F_4 [Fau99] algorithm, then, it is converted into a lexicographic one using the seminal FGLM algorithm or its faster variant, in generic cases, sparse-FGLM [FM17]. We refer to [BES21] for a complete description and implementation of this framework in the open-source C library msolve. Assuming the system has k solutions, sparse-FGLM relies on the Wiedemann algorithm, or on its faster version block-Wiedemann [Hyu+20]. Its bottleneck is the computation of $\frac{2k}{n}$ matrices $v_0, v_1 = Mv_0, \ldots, v_{\frac{2k}{n}-1} = Mv_{\frac{2k}{n}-2}$, where M is a $k \times k$ matrix given by the first Gröbner basis and v_0 is $k \times n$ and random. Typically, n is a small power of 2 such as, usually, 32. The matrix M has a special structure, each column is either made of only zeroes and one 1 (like a column of the identity matrix) or is dense. Furthermore, the asymptotics of m, the number of dense columns, has been thoroughly studied in the generic case in [FM17]. These products yield a complexity $O(mk^2)$.

When these systems are over $\mathbb{Q}$, the growth of the coefficients is controlled through a multi-modular approach. As a consequence, this abstract only deals with systems over a finite field $\mathbb{F}_p \simeq \mathbb{Z}/p\mathbb{Z}$ of size and characteristic a prime number p.

The goal of this work is to develop matrix multiplication algorithms over finite fields for GPU architectures. Indeed, GPUs are, by design, well-suited to process large blocks of data in parallel and thus perform linear algebra routines, more so than CPUs. However, current GPUs natively handle double-precision floating-point number arithmetic but only simulate long integer ones through short integer arithmetic, which comes with an overhead. For instance, Nvidia CUDA and Tensor cores, do not natively support 64-bit integer types while they do for 64-bit floating-point types. Hence, we need to do exact arithmetic over finite fields with floating-point types. This approach has been explored in CPU-only libraries: FFLAS [gro19], NTL [Sho21], FLINT [HJP13], Mathemagix [HLQ17].

Contributions. Our contribution is twofold. First, we have developed a GPU implementation of matrix multiplication over finite fields with techniques borrowed from the aforementioned CPU-only libraries. However, the multi-modular approach requires to have large primes of size at least 30 bits, yet these existing algorithms cannot handle primes with more than 26 bits and performance drops significantly when approaching this limit. Therefore, our second contribution is to propose a multiword algorithm to remove this limitation and alleviate the performance drop. In the future, we aim to integrate these advances in the msolve [BES21] library.

2 Algorithms

2.1 Floating-point dot product and reduction

Computing exactly with finite fields elements using floating-point types requires defining a modulo operator similar to integer types. In this work, we use the finite field reduction algorithm described in [HLQ17, Sec. 3.3, Function 16]. This algorithm leverages the fused multiply-add (FMA) instruction to reduce modulo p an integer that can be stored exactly in a double precision floating-point.

These reductions are still costly. In the FFLAS [DGP08] library, the authors lower the number of reductions in the dot product by reducing after partial dot products of size $\lambda = \left\lfloor \frac{2^\ell}{(p-1)^2} \right\rfloor$ where ℓ is the mantissa bitsize of our floating-point type (53 for double-precision arithmetic).

2.2 Block-product algorithm

Algorithm 1 relies on the aforementioned dot product to multiply $A \in \mathbb{F}_p^{m\times k}$ and $B \in \mathbb{F}_p^{k\times n}$ using only floating-point operations. This product is equal to the sum of the submatrices products A_jB_j, where $A_j \in \mathbb{F}_p^{m\times\lambda}$ and $B_j \in \mathbb{F}_p^{\lambda\times n}$. The product A_jB_j is stored in a buffer T (line 4) of size mn and then each of its coefficients is reduced with the FMA modular reduction (line 6). Finally, the buffer is added in the resulting matrix C (line 8).

Additional reductions when adding the buffer are necessary only if $(p-1)\left\lceil \frac{k}{\lambda} \right\rceil$ is larger than 2^ℓ. Approximately, we obtain this upper bound: $(p-1)^3k \leq 2^{2\ell}$. Extra reductions become mandatory no matter the value of k when p is at least 35 bits.

The separation of modular reductions and floating-point operations enables the use of an efficient routine for the floating-point matrix multiplication, namely the cuBLAS `dgemm` [NVI]. This makes Algorithm 1 particularly attractive for GPU architectures. The floating-point matrix multiplication requires $2mkn$ floating-point operations and there are only $mn\left\lceil\frac{k}{\lambda}\right\rceil$ reductions. We can thus hope that the performance of the algorithm is mainly determined by the performance of the BLAS, which is in turn usually very close to the maximum theoretical performance of the hardware.

Algorithm 1: λ-block matrix product over $\mathbb{F}_p$

```
Input  : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$ stored in double
         precision;
         such that $a_{i,j}, b_{i,j} < 2^{26}$;
         $p$, the characteristic of $\mathbb{F}_p$;
         $\lambda$, modular reduction delay.
Output: $C = AB \in \mathbb{F}_p^{m\times n}$ stored in double precision.
1  def FFMatMulSW:
2      $C = 0 \in \mathbb{F}_p^{m\times n}$
3      for $j = 1$ to $\lceil k/\lambda \rceil$ do
4          $T = A_j * B_j$     // Submatrices of sizes
5                              // $m \times \lambda$ and $\lambda \times n$
6          $T = T \bmod p$     // Modular reduction with
7                              // FMA
8          $C = C + T$         // Reductions here only if
9                              // $\lceil k/\lambda \rceil p \geq 2^{53}$
10     end
11     return $C \bmod p$
```

Algorithm 2: Multiword matrix multiplication

```
Input  : $A \in \mathbb{F}_p^{m\times k}$, $B \in \mathbb{F}_p^{k\times n}$ stored
         in double precision;
         $p$ characteristic of $\mathbb{F}_p$.
Output: $C = AB \in \mathbb{F}_p^{m\times n}$ stored in
         double precision.
1  def FFMatMulMW:
2      $\lambda = 2^{52-t}$
3      $(A_h,\ A_l) =$ MW-Decomposition$(A)$
4      $(B_h,\ B_l) =$ MW-Decomposition$(B)$
5      $M_1 =$ FFMatMulSW$(A_h, B_h, p, \lambda)$
6      $M_2 =$ FFMatMulSW$(A_h, B_l, p, \lambda)$
7      $M_3 =$ FFMatMulSW$(A_l, B_h, p, \lambda)$
8      $M_4 =$ FFMatMulSW$(A_l, B_l, p, \lambda)$
9      $M_3 = M_3 + M_2$
10     $M_3 = (2^{t/2} \cdot M_3) \bmod p$
11     $M_1 = (2^t \cdot M_1) \bmod p$
12     return $M_1 + M_3 + M_4 \bmod p$
```

2.3 Multiword algorithm

Algorithm 1 is limited to primes p not exceeding 26 bits because we need twice as many bits to store the products of coefficients of A and B in the double precision buffer T (line 4). Yet, we require a matrix product that can handle 30-bit prime fields to solve polynomial systems. We have developed a multiword approach described in Algorithm 2 that raises the limit to 35 bits, and potentially more at the expense of some additional modular reductions (line 8 in algorithm 1).

Algorithm 2 uses more than one double precision machine word to represent exactly the results of the product AB. First, it splits each matrix A and B into two matrices A_h, A_l and B_h, B_l, respectively, containing each the quotient (high part) and the remainder (low part) of the division of the initial matrix by $2^{t/2}$ where t is the bitsize of the prime p (lines 3–4). The product AB is then expressed as $C = (2^{t/2}A_h + A_l) \cdot (2^{t/2}B_h + B_l) = 2^t A_h B_l + 2^{t/2}(A_h B_l + A_l B_h) + A_l B_l$ (lines 5–8).

Compared with the single word Algorithm 1, Algorithm 2 thus requires three extra subproducts. However, the λ block size is larger for each of these products, which thus requires fewer reductions.

3 GPU benchmarks

We now present benchmarks of Algorithms 1 (block product) and 2 (multiword) for computing a matrix product AB on a single Nvidia Ampere GPU (model A40). We use matrices of dimensions $m = 15000$, $k = 45000$ and $n = 32$, which are typical dimensions for the sparse-FGLM algorithm using block-Wiedemann.

We measure the performance Π of each algorithm in GFLOPS (Giga floating-point operation per second) using the formula $\Pi = \frac{2mkn}{10^9\tau}$, where τ is the runtime of the algorithm, and where $2mkn$ represents the number of flops required to multiply $m \times k$ and $k \times n$ matrices by a floating-point product. Note that we use this formula regardless of the actual number of flops performed by each algorithm.

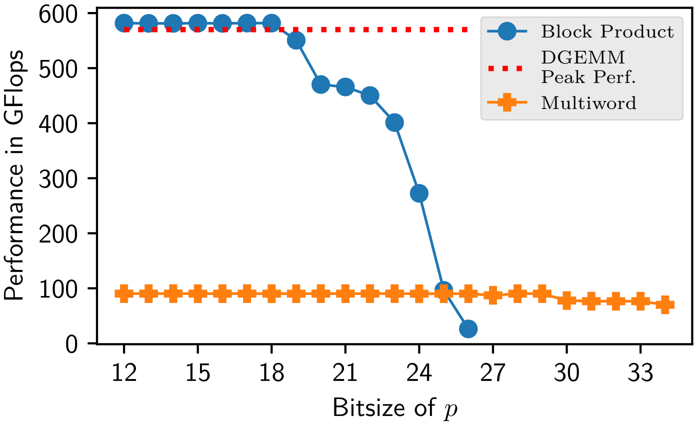

Figure 1: Performance of block and multiword matrix product on an A40 GPU ($m = 15000$, $k = 45000$, $n = 32$).

Figure 1 plots the result of our performance benchmark. The performance of Algorithm 1 is constant for p with between 12 and 18 bits since the limiting factor is the `dgemm` performance (red-dotted line), which is near the peak performance of the hardware (584 GFLOPS). The block size λ reduces as the prime size increases, which explains why the performance of Algorithm 1 drops for primes p with between 23 and 26 bits. In contrast, the λ used by the multiword algorithm is larger and as a result, its performance is almost constant, with only a slight reduction starting from 30-bit primes. The multiword algorithm is currently more than four times slower than the single-word algorithm; this difference is under investigation and we expect to be able to improve its performance in future implementations.

References

[BES21] J. Berthomieu, Ch. Eder, and M. Safey El Din. "Msolve: A Library for Solving Polynomial Systems". In: *Proceedings of ISSAC'21*. ACM, 2021, pp. 51–58. DOI: 10/gk8549.

[DGP08] J.-G. Dumas, P. Giorgi, and C. Pernet. "Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages". In: *ACM T. Math. Software* 35.3 (2008), pp. 1–42. DOI: 10/dj6zp4.

[Fau99] J.-Ch. Faugère. "A New Efficient Algorithm for Computing Gröbner bases (F4)". In: *J. Pure Appl. Algebra* 139.1 (1999), pp. 61–88. DOI: 10/bpq5dx.

[FM17] J.-Ch. Faugère and Ch. Mou. "Sparse FGLM algorithms". In: *J. Symb. Comput.* 80.3 (2017), pp. 538–569. DOI: 10/gfz47c.

[gro19] The FFLAS-FFPACK group. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package*. v2.4.1. http://github.com/linbox-team/fflas-ffpack. 2019.

[HJP13] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. Version 2.4.0, http://flintlib.org. 2013.

[HLQ17] J. van der Hoeven, G. Lecerf, and G. Quintin. "Modular SIMD arithmetic in Mathemagix". In: *ACM T. Math. Software* 43.1 (2017), pp. 1–37. DOI: 10/f82vvw.

[Hyu+20] S. G. Hyun et al. "Block-Krylov techniques in the context of sparse-FGLM algorithms". In: *J. Symbolic Comput.* 98 (2020), pp. 163–191. DOI: 10/gkx9.

[NVI] NVIDIA. *cuBLAS documentation*. https://docs.nvidia.com/cuda/cublas/#.

[Sho21] V. Shoup. *NTL: a library for doing numbery theory*. 2021. URL: http://www.shoup.net.

The authors are supported by the joint ANR-FWF ANR-19-CE48-0015 ECARP project, the ANR grants ANR-18-CE33-0011 Sesame, ANR-19-CE40-0018 De Rerum Natura and ANR-20-CE48-0014 NuSCAP projects and grant FA8665-20-1-7029 of the EOARD-AFOSR. We thank the referees for their valuable comments on the paper.