Implementation Techniques For Fast Polynomial Arithmetic In A High-level Programming Environment

Akpodigha Filatei†, Xin Li†, Marc Moreno Maza†, and Éric Schost⋆
⋆: LIX, École polytechnique, 91128 Palaiseau, France.
†: ORCCA, University of Western Ontario, London, Canada.
CAIMS-MITACS, June, 2006.

Introduction

It is well known that symbolic computations may incur high time and space consumption. Hence, designing and implementing high-performance algorithms in this field is a challenge. Controlling expression swell and speeding up efficiency critical routines is a clear need: This is the goal of the so-called modular algorithms and fast arithmetic operations [4].
Modular algorithms is an active and successful research area. Fast arithmetic operations are known since the 1960's but very few symbolic computation software implement them until today. In fact, it was believed that they were not relevant in practice. In the last decade, however, successful implementations of fast arithmetic operations were realized in middle or low-level programming languages [5,6].
Achieving high-performance with high-level programming environments is also desirable for the working mathematician: they help reducing development cycles and validating code; they provide important features like generic programming. The cost of this convenience is often less efficient code.
AXIOM [2] and *Aldor* [1] are such environments that were designed for expressing mathematical properties and algorithms in a powerful manner. They are the frameworks that we have chosen to develop implementation techniques for fast arithmetic in a high-level programming environment [3].

Main Results

We distinguished two cases: the *generic case* and the *non-generic case*; we investigated them using *Aldor* and *AXIOM* respectively.
The generic case: In *Aldor* we developed generic code for fast algorithms, such as *Fast Fourier Transform* (*FFT*)-based univariate polynomial multiplication for an arbitrary field of coefficients.
The non-generic case: In *AXIOM* we focused on dense polynomials over finite fields, since they are central in modular methods.
We developed highly efficient implementation of efficiency-critical operations, *FFT*-based univariate polynomial multiplication and power series inversion.
In the generic case, our code is comparable (slower by a factor between 2 and 3, see Figures [7,8]) with the best known packages, *NTL* [6] and *Magma* [5], which are non-generic implementations in lower-level languages (*C*, *C++*). **In the non-generic case**, our code is comparable, and often better, than that of *NTL* and *Magma*, see Figures [2,7].

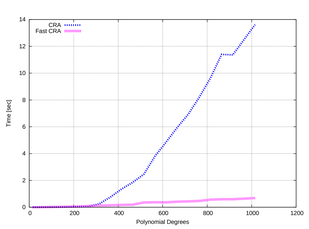

Figure-1

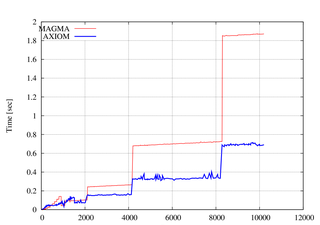

Figure-2

Based on these, we translated *from the book* other fast algorithms: (1) in *Aldor*, the Fast Extended Euclidean Algorithm (Fast *EEA*) and the Fast Chinese Remaindering Algorithm (Fast *CRA*), (2) in *AXIOM*, multivariate polynomial multiplication using Kronecker's trick. Figure [1] compares our *Aldor* implementations of the Standard and Fast *CRA*'s. Figure [2] compares our *AXIOM* bivariate polynomial multiplication to that of *Magma*. These benchmarks illustrate the fact that we have developed a framework for implementing fast algorithms in high-level programming environments.

Fast Polynomial Arithmetic

Fast algorithms for polynomial arithmetic are algorithms:

- for operations such as multiplication, division, GCD, CRA,
- with a running time complexity quasi-linear.

Figure [1] shows a comparison between a fast and a standard algorithm for the *CRA*. Fast algorithms are challenging to implement. They usually require change of data representation (e.g. for the *FFT*-based multiplication) and have large constant in their complexity estimates.

The Generic Case

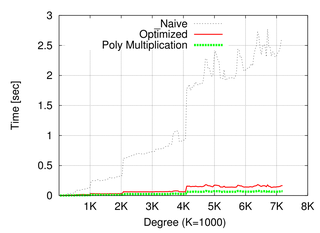

Figure-3

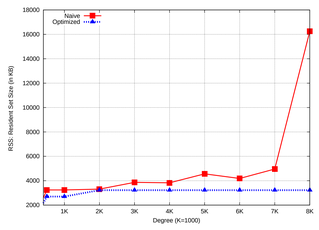

Figure-4

We write optimizer-friendly and garbage collector (GC)-friendly code without compromising the high-level nature of our implementations.
Figures [3] and [4] compare running time and memory consumption for different implementations of power series inversion, a critical operation in the fast *EEA* [4]. On both figures, the upper curve corresponds to a direct implementation, whereas the curve below corresponds to our optimized code.

The Non-Generic Case

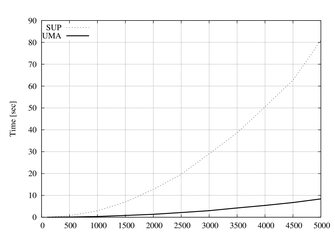

Figure-5

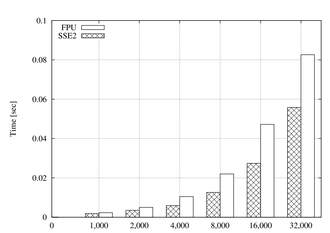

Figure-6

Obtaining fast implementations of algorithms over finite fields (such as $\mathbb{Z}/p\mathbb{Z}$, for a prime p) requires low-level considerations of data structures, machine arithmetic, memory traffic, compiler optimization. *AXIOM* is an ideal environment for a multi-level language implementation strategy.
In Figure [5]: (1) the upper curve corresponds to classical univariate polynomial multiplication (modulo a prime integer) implemented in the *AXIOM* programming language (2) the lower curve is our *Assembly* implementation for the same operations **called** from the *AXIOM* level.
Figure [6] shows the importance of architecture-aware implementations with two implementations of *FFT*-based multiplication: in the case of the dark columns, *SSE2* registers are used, whereas for the white columns, we rely on "traditional" *FPU* registers.

Benchmarks

- We compared our implementations with *NTL* [6] and *Magma* [5].
- Figure [7] shows timings for the *FFT*-based univariate multiplication.
- Figure [8] reports on polynomial GCD computations via the Fast *EEA*.
- More experiments are reported in [3].
- Further goal: a highly efficient polynomial systems solver written in a high-level programming environment.

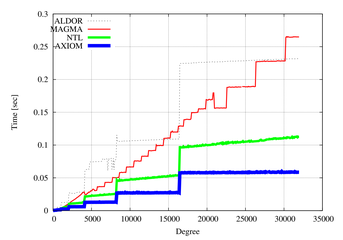

Figure-7

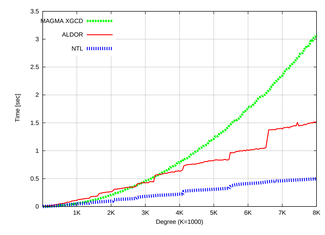

Figure-8

References

[1] *Aldor* web site: `http://www.aldor.org`.

[2] *AXIOM* web site: `http://page.axiom-developer.org`.

[3] A. Filatei, X. Li, M. Moreno Maza and É. Schost, Implementation Techniques for Fast Polynomial Arithmetic in a High-level Programming Environment, Proc. ISSAC'06, ACM Press, 2006.

[4] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 1999.

[5] The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney, *Magma*, `http://magma.maths.usyd.edu.au/magma/`.

[6] V. Shoup, *NTL*: A Library for doing Number Theory, `http://www.shoup.net/ntl/`.