

# MBLAS: Modular Basic Linear Algebra Subprograms, Design and Speedup Techniques

Yasuhiro KAWAME

Hirokazu MURAO

Department of Computer Science, University of Electro-Communications

## 1 Introduction

Computation in prime fields  $\mathbb{Z}_p$  is used by various algorithms in computer algebra, and plays an important role in practice, especially when  $p$  is represented direct by machine datatype. The application area is wide spreaded, but in most cases, major part of the calculation can be reduced to a combination of basic linear operations, as in the calculations with polynomials over  $\mathbb{Z}_p$ . Arithmetic in  $\mathbb{Z}_p$  is very simple and the reduction by mod  $p$  costs significantly. To achieve practical superior efficiency, detailed analysis and careful tuning is required, especially when treating vectors and matrices, a bulk of homogeneous data. With this recognition and the necessity for sharable and reusable efficient subprograms in mind, we started a few years ago a project to develop a program library of modular version of BLAS, basic linear algebra subprograms. This poster gives an overview of our design, and describes some practical and successful techniques used in our implementation, as well as showing experimental results. There is a similar software by Dumas *et al.* [1]. Our effort will be characterized by the following features.

- Definition of functionalities specific to and required by computer algebra, which is different from numerical processing.
- Multiple datatypes with different precisions (8/16/32-bit) for space and time efficiency.
- Table-driven mod  $p$  reduction.
- Use of vector processing in integer arithmetics, with currently Intel's SSE2 and full-scale vector processor in scope as machine target.

## 2 Overview of MBLAS

**(Representation)** Main target of our implementation is 32-bit machines. For space efficiency, we treat multiple datatypes of different precisions, and we represent elements of  $\mathbb{Z}_p$  by 8/16/32-bit integers. One more alternative will be the use of floating-point numbers. However, we cannot find out any merit of this choice, except when used for dedicated processors to numerical processing, and decided not to use in the current version. One more method for representation might be the use of Jacobi logarithm [3], as reported in [1]. It seems too large-scaled only for the calculations in prime field, and not used in our current implementation.

**(Functionalities)** MBLAS defines a set of functionalities for computer algebra, as follows.

- (level 1) exchange, copy, negation, additive operations, scalar multiplication, inner product of vectors. basic matrix transformation for computing matrix standard forms.
- (level 2 & 3) matrix addition and subtraction, matrix-vector multiplication, vector outer product, matrix multiplication with naive and fast algorithms, and their combinations.

For most funtions, both inplace-type and copy-type routines are prepared separately.

### 3 Arithmetics in $\mathbb{Z}_p$

We represent elements of  $\mathbb{Z}_p$  by non-negative integers  $< p$ , and store them as unsigned 8/16/32-bit integer.

With respect to the additive operations, division is not required for mod  $p$ , and can be replaced by subtraction or addition by  $p$  for the out-of-range results ( $\geq p$  or  $< 0$ ). Overflow may occur while addition or subtraction, and complicates the detection of out-of-range results. If  $p$  is small enough to guarantee no overflow, we can simplify the condition of this detection. When multiple data are treated simultaneously as in vectors or matrices, check of the magnitude of  $p$  should be done only once for all data. There are two kinds of multiplicative operations: (1) *admul* type as  $ab + c \bmod p$  for  $a, b, c \in \mathbb{Z}_p$ , and (2) *inner-product* type as  $\sum_i a_i b_i \bmod p$  where  $a_i, b_i \in \mathbb{Z}_p$ . In both cases, intermediate expressions require at least double  $p$ 's precision, but treatment towards mod  $p$  is somewhat different. In the former case, smaller value of  $ab + c$  is preferred to simplify this reduction, while in the latter case, as many  $a_i b_i$ 's are accumulated as possible so far as the sum fits in variable's precision of the intermediate expression to reduce the number of (mod  $p$ )-reductions performed.

#### Efficient mod $p$ reduction – use of tables

There are a few efficient methods for computing mod  $p$ . One method is to precompute the value of  $1/p$  as floating-point number with sufficient precision and use it for determining the quotient of  $A \bmod p$ , which can eliminate costly division. This is most effective when all data are treated as reals, because datatype conversion between real and integer costs very much.

Another method, which we propose here, is the use of precomputed tables. We partition integer data of dividends into multiple bit-sequences, and we prepare a table of (mod  $p$ ) values for every possible bit-pattern of each bit-sequence. We expect that taking wider bit-sequence (bigger table) fastens the (mod  $p$ ) operations, and our question is what is the appropriate value. We tested three cases of partitioning. Our experiment indicated the partitioning into  $4 \times 8$ -bit is most efficient both in time and space, and has shown 2 ~ 3 times speedup compared with other methods.

#### Vector processing – use of SSE2

To speedup integer arithmetics, we apply vector processing. Current version implements only Intel's SSE2[2], and also a large-scale vector processor is in our scope and will be a target in future.

With SSE2, multiple 8/16/32/64-bit integer data in 128-bit vector can be treated simultaneously, and additive and multiplication operations are supported. Our implementation is done via intrinsics, and therefore we must check what type of operations are supported. The problem with SSE2 is the integer comparisons, which is supported only for signed integers. We must be careful about the overflow check, and have obtained simple method for case distinctions. This distinction is simple if  $p$  is sufficiently small, and this check ought to be done only once for all data in vectors or matrices. We observed speedup with ratio up to 3 in very successful cases. More details of our experimental results will be reported in the poster.

### References

- [1] DUMAS, J. G., GAUTIER, T., AND PERNET, C. Finite field linear algebra subroutines. In *Proc. ISSAC '02* (2002), ACM Press, pp. 63–74.
- [2] INTEL CORP. *IA-32 Intel Architecture Software Developer's Manual*.
- [3] LIDL, R., AND NIEDERREITER, H. *Finite Fields*, vol. 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge Univ. P., 1997.