Rational Number Reconstruction Using Chinese Remainder Theorem on GPU

Toru Fukaya
Graduate School of Engineering and Science
Shibaura Institute of Technology, Saitama, Japan.
Email: mf16056@shibaura-it.ac.jp

Tomoyuki Idogawa
College of Systems Engineering and Science
Shibaura Institute of Technology, Saitama, Japan.
Email: idogawa@sic.shibaura-it.ac.jp

***Abstract*—The purpose of this study is to make rational number arithmetic fast. For this purpose, we implemented a rational number reconstruction method, a kind of modular algorithms, in which we used Chinese remainder theorem in order to parallelize calculation. We implemented it on CPU and GPU. Then, we applied them to some examples of computing such as inner products, Frobenius normal forms of matrices and determinants of matrices to examine their efficiencies. As a result, we showed that our implementations calculated faster than the standard arithmetic by using GMP at least in the latter two cases (i.e., computing of Frobenius normal forms and determinants). We also showed that the GPU version calculated 8.3 times faster at most than the CPU version.**

1. Introduction

Mostly, numerical calculations are done by using floating point numbers. There are great benefits because floating point operations can be done very fast nowadays, but sometimes there arise some problems because of round-off errors which floating point systems essentially have. To avoid such problems, one would use quadruple precision or interval arithmetic systems.

On the other hand, there are computer algebra systems in which each expression is processed symbolically. In these systems, rational number representations are used for coefficients of polynomials or rational expressions since they can be calculated exactly. They are useful to computer aided proof because of their strictness. Moreover, they can be used to estimate errors caused by floating point systems. In such a sense, the rational number representations are very powerful, but they have big drawback that the calculations are rather slow.

In some cases, the coefficients of intermediate results during calculation are much larger than those of the final result. In such a calculation, it takes very long time and requires very large memory space. In these cases, using a rational reconstruction method based on modular algorithms, one can expect that it needs rather smaller memory space and does not decrease the efficiency so much.

Regarding the efficiency, GPU (Graphics Processing Unit) has been attracted by its high performance in computing, recently. It is reported that a certain problem was calculated by GPU 100 times faster than by CPU. There is also a study to accelerate the multiprecision arithmetics [3].

In this study, We implemented the modular algorithm to accelerate rational number arithmetic and evaluated it's efficiency by some numerical experiments. In our implementation, we used Chinese remainder theorem to be able to parallelize the calculations (see §2.4). In addition to CPU, we implemented the algorithm on GPU which is good at parallel computing to accelerate the calculations much more.

2. Modular algorithm

In this section, we introduce the modular algorithms. Using a modular algorithm, one can avoid problems caused by intermediate expression swell. Further, using the small primes modular algorithm, one can expect to speed up calculations much more.

Instead of solving an algebraic computation problem over a Euclidean domain $\mathbf{R}$ directly, modular algorithms solve it modulo one or several ideals and reconstruct from this (these) result(s) the desired solution in $\mathbf{R}$ to the original problem. There are three variants of a modular algorithm, those use a (big) prime modulus, prime power modulus or some (small) prime moduli, respectively. A concept of the simplest one, which uses a (big) prime modulus, is illustrated in Fig. 1. The modular algorithm calculates along the dashed line on the figure. In this study, we adopt one which uses some prime moduli, illustrated in Fig. 2 (both Fig. 1 and 2 were drawn with referring to [2]).

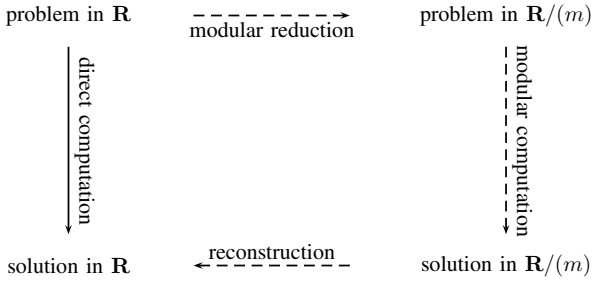

Figure 1. General concept of modular algorithms.

978-1-5090-0806-3/16/$31.00 copyright 2016 IEEE
ICIS 2016, June 26-29, 2016, Okayama, Japan

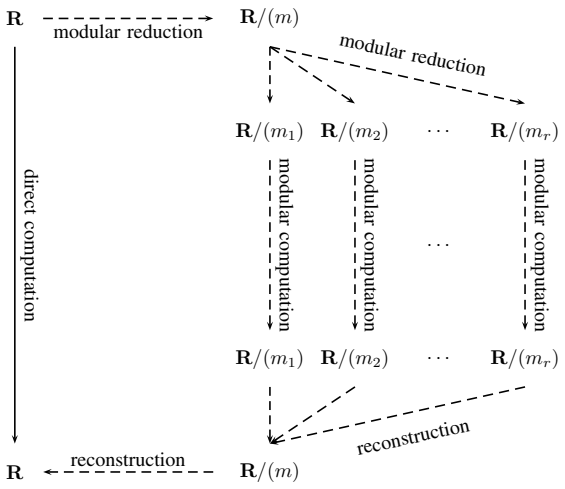

Figure 2. A concept of small primes modular algorithms.

2.1. The extended Euclidean algorithm

In modular algorithms, the following Extended Euclidean Algorithm (EEA) is one of the most important algorithms. Actually, EEA has various applications, e.g., to solve a linear indeterminate equation or a linear congruence equation, a division in $\mathbb{Z}_m$, a rational number reconstruction and so on.

EEA is shown in Alg. 2.1 (see [2, p.47]), where each symbol denotes an integer.

Algorithm 2.1 Extended Euclidean Algorithm

Require: f, g $(f \geq g \geq 0)$.
Ensure: l, $\{r_i\}_{i=0}^{l+1}$, $\{s_i\}_{i=0}^{l+1}$, $\{t_i\}_{i=0}^{l+1}$, $\{q_i\}_{i=1}^{l}$.
1: $(r_0, s_0, t_0) \leftarrow (f, 1, 0)$
2: $(r_1, s_1, t_1) \leftarrow (g, 0, 1)$
3: $i \leftarrow 1$
4: **while** $r_i \neq 0$ **do**
5: $q_i \leftarrow \lfloor r_{i-1}/r_i \rfloor$
6: $\begin{pmatrix} r_{i+1} \\ s_{i+1} \\ t_{i+1} \end{pmatrix} \leftarrow \begin{pmatrix} r_{i-1} \\ s_{i-1} \\ t_{i-1} \end{pmatrix} - q_i \begin{pmatrix} r_i \\ s_i \\ t_i \end{pmatrix}$
7: $i \leftarrow i + 1$
8: **end while**
9: $l \leftarrow i - 1$
10: **return** l, $\{r_i\}_{i=0}^{l+1}$, $\{s_i\}_{i=0}^{l+1}$, $\{t_i\}_{i=0}^{l+1}$, $\{q_i\}_{i=1}^{l}$

2.2. Chinese remainder algorithm

The following Chinese Remainder Theorem (CRT) is another key algorithm for small primes modular algorithms. Alg. 2.2 solve the simultaneous congruence equations in the theorem, efficiently (see [1, p.290]).

Theorem 2.1 (Chinese Remainder Theorem)**.** *Let $m_1, m_2, \ldots, m_r \in \mathbb{Z}$ be pairwise coprime. Then, any given $u_1, u_2, \ldots, u_r \in \mathbb{Z}$, there exists $u \in \mathbb{Z}$ such that*

$$\begin{aligned} u &\equiv u_1 \pmod{m_1}, \\ u &\equiv u_2 \pmod{m_2}, \\ &\vdots \\ u &\equiv u_r \pmod{m_r}. \end{aligned}$$

The solution u is unique in the sense of modulo $M = m_1 m_2 \cdots m_r$.

Algorithm 2.2 Chinese Remainder Algorithm

Require: m_i pairwise coprime, u_i $(i = 1, 2, \ldots, r)$,
Ensure: u
1: For all $1 \leq i < j \leq r$, solve $c_{ij} m_i \equiv 1 \pmod{m_j}$ for c_{ij}.
2: **for** $j = 1, \ldots, r$ **do**
3: $v_j \leftarrow u_j \bmod m_j$
4: **for** $i = 1, \ldots, j-1$ **do**
5: $v_j \leftarrow (v_j - v_i) c_{ij} \bmod m_j$
6: **end for**
7: **end for**
8: $u \leftarrow v_r$
9: **for** $i = r-1, \ldots, 1$ **do**
10: $u \leftarrow u m_i + v_i$
11: **end for**
12: **return** u

2.3. Rational number reconstruction

The rational number reconstruction algorithm is based on the following theorem 2.2.

Theorem 2.2 (Rational Number Reconstruction)**.** *Let m, n be positive integers. If $\frac{a}{b} \in \mathbb{Q}$ satisfies*

$$|a| < n,\ 0 < b < n,\ \gcd(a, b) = 1, \tag{1}$$

then the congruence equation $a \equiv bu \pmod{m}$ has a solution $u \in \mathbb{Z}$ if and only if $\gcd(b, m) = 1$. When the equation has a solution, it is unique in the sense of modulo m. Conversely, for given m, n, u, there exists at most one rational number $\frac{a}{b}$ which satisfies $a \equiv bu \pmod{m}$ when $m \geq 2n^2$.

The algorithm which reconstructs a rational number $\frac{a}{b}$ from m, n and u is shown in Alg. 2.3 (see [2, §5.10]).

2.4. Rational number reconstruction using CRT

Combining Alg. 2.2 and 2.3, we get the rational number reconstruction algorithm using Chinese remainder theorem (Alg. 2.4, which was made with referring to [2, p.129]). Note that the computation in each modulo (Step 8 of Alg. 2.4) can be done in parallel.

Algorithm 2.3 Rational Number Reconstruction

Require: m, n, u $(m \geq 2n^2)$.
Ensure: (a, b), $a \equiv bu \pmod{m}$.
1: Calculate Alg. 2.1 for m, u. Its results are l, $\{r_i\}_{i=0}^{l+1}$, $\{s_i\}_{i=0}^{l+1}$, $\{t_i\}_{i=0}^{l+1}$, $\{q_i\}_{i=1}^{l}$.
2: Find j which satisfies $r_j < n \leq r_{j-1}$.
3: **if** $j \leq l$ **then**
4: Find q which satisfies $r_{j-1} - qr_j < n \leq r_{j-1} - (q-1)r_j$.
5: **else**
6: $q \leftarrow 0$
7: **end if**
8: $r_j^* \leftarrow r_{j-1} - qr_j$, $t_j^* \leftarrow t_{j-1} - qt_j$
9: **if** $|t_j| < n$ **then**
10: **return** $(\mathrm{sign}(t_j)r_j, \mathrm{sign}(t_j)t_j)$
11: **else if** $|t_j^*| < n$ **then**
12: **return** $(\mathrm{sign}(t_j^*)r_j^*, \mathrm{sign}(t_j^*)t_j^*)$
13: **else**
14: No solution.
15: **end if**

Algorithm 2.4 Rational Number Reconstruction Using CRT

Require: f : k-variables rational function and canonical rational numbers $\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_k}{b_k}$.
Ensure: $\frac{a}{b} = f\left(\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_k}{b_k}\right)$.
1: Prepare $m_i \in N$, $m_i \geq 2$ $(1 \leq i \leq r)$, $\gcd(m_i, m_j) = 1$ $(i \neq j)$ for next calculations.
2: **for** $i = 1, \ldots, r$ **do**
3: **for** $j = 1, \ldots, k$ **do**
4: $x_{ij} \leftarrow b_j^{-1} a_j$ (in $\mathbb{Z}_{m_i}$)
5: **end for**
6: **end for**
7: **for** $i = 1, \ldots, r$ **do**
8: $u_i \leftarrow f(x_{i1}, x_{i2}, \ldots, x_{ik})$
9: **end for**
10: Do Alg. 2.2 for u_i, m_i in order to find u.
11: Do Alg. 2.3 u, $m = \prod_{i=1}^{r} m_i$, $n = \lfloor \sqrt{m/2} \rfloor$ in order to find (a, b).
12: **return** $\frac{a}{b}$

Here we will show an example. The objective is to calculate the rational expression

$$\left(\frac{1}{2}\right)^3 - \left(\frac{2}{3}\right)^3 - \left(\frac{5}{6}\right)^3.$$

Let $f(c_1, c_2, c_3) = c_1^3 - c_2^3 - c_3^3$.

Step 1. Prepare $m_1 = 13$, $m_2 = 17$.

Step 2∼9.

Calculation in $\mathbb{Z}_{13}$	Calculation in $\mathbb{Z}_{17}$
$1 \cdot 2^{-1} = -6,$	$1 \cdot 2^{-1} = -8,$
$2 \cdot 3^{-1} = 5,$	$2 \cdot 3^{-1} = -5,$
$5 \cdot 6^{-1} = 3.$	$5 \cdot 6^{-1} = -2.$
$(-6)^3 + 5^3 + 3^3 = 9.$	$(-8)^3 + (-5)^3 + (-2)^3 = 12.$

Step 10. We get $u \equiv 9 \pmod{13}$, $u \equiv 12 \pmod{17}$ and hence $u \equiv 165 \pmod{221}$, where $221 = 13 \cdot 17$.

Step 11.

r_i	q_i	s_i	t_i
221		1	0
165	1	0	1
56	2	1	-1
53	1	-2	3
3		3	$-$**4**

Step 12. Finally, we get the result $-\frac{3}{4}$.

3. Algorithms for more fast calculation

One may avoid intermediate expression swells and speed up calculations of rational expression by implementing Alg. 2.4. In addition to Alg. 2.4, we implemented the following algorithms to accelerate calculations much more.

3.1. Montgomery multiplication

At first, we introduce Montgomery multiplication [6] which speeds up modular multiplications. Using Montgomery reduction MR defined below, one can calculate $c = ab \bmod N$ by

$$c \leftarrow \mathrm{MR}(\mathrm{MR}(ab)R_2),$$

where $R_2 = R^2 \bmod N$.

Definition 3.1 (Montgomery reduction)**.** *Given integers N, R and R' which satisfy $2 \leq N < R$, $\gcd(N, R) = 1$ and $RR' \equiv 1 \pmod{N}$, Montgomery reduction $\mathrm{MR} : \mathbb{Z} \to \mathbb{Z}$ with R and modulo N is defined by*

$$\mathrm{MR}(T) = TR' \bmod N \quad (0 \leq T < NR).$$

The value of Montgomery reduction $\mathrm{MR}(T)$ can be calculated by Alg. 3.1. When one choose $R = 2^d$, operations $\cdot \bmod R$ and $\cdot / R$ can be replaced by bitwise AND and bit shifting operations, respectively. Hence one can avoid divisions those are required in $\bmod N$ operations and cost rather high in computing.

3.2. Lehmer's GCD algorithm

When the Extended Euclidean Algorithm (EEA) is applied to big multiprecision integers, there occurs division operations between multiprecision integers many times. These operations cost rather high in computing. But if one calculates only by upper few digits, then one may decrease the number of multiprecision operations.

Algorithm 3.1 Montgomery Reduction

Require: integers R, N s.t. $2 \leq N < R$, $\gcd(R, N) = 1$,
integer $R' \in [0, N-1]$ s.t. $RR' \equiv 1 \pmod{N}$,
integer $N' \in [0, R-1]$ s.t. $NN' \equiv -1 \pmod{R}$,
integer $T \in [0, NR-1]$.
Ensure: integer $t \in [0, N-1]$ s.t. $t \equiv TR' \pmod{N}$.

$t \leftarrow (T + (TN' \bmod R)N)/R$.
if $t \geq N$ **then**
  return $t - N$
else
  return t
end if

For example, let consider applying EEA to 8-digit natural numbers $u = 31415926$ and $v = 20000000$. Then, choose 4-digit natural numbers u', v', u'' and v'' such that

$$\frac{u'}{v'} < \frac{u}{v} < \frac{u''}{v''}. \tag{2}$$

Here, let us use $u' = \lfloor u/10^4 \rfloor = 3141$, $v' = \lfloor v/10^4 \rfloor + 1 = 2001$, $u'' = \lfloor u/10^4 \rfloor + 1 = 3142$, $v'' = \lfloor v/10^4 \rfloor = 2000$. Then, apply EEA to the pair (u', v') and the other (u'', v''), respectively, until their quotients q_i differ (see Tab. 1 and 2). It can be shown that the processes (q_i, s_i and t_i) of EEA for (u', v'), (u'', v'') and moreover (u, v) are exactly same while q_i's of (u', v') and (u'', v'') are same. Therefore, one can say that EEA for (u, v) will go on such as Tab. 3 in this example.

TABLE 1. EEA FOR (u', v').

r_i	q_i	s_i	t_i
3141		1	0
2001	1	0	1
1140	1	1	-1
861	1	-1	2
279	3	2	-3
24	11	-7	11

TABLE 2. EEA FOR (u'', v'').

r_i	q_i	s_i	t_i
3142		1	0
2000	1	0	1
1142	1	1	-1
858	1	-1	2
284	3	2	-3
6	47	-7	11

↓

TABLE 3. EEA FOR (u, v).

r_i	q_i	s_i	t_i
u		1	0
v	1	0	1
$u - v$	1	1	-1
$-v + 2v$	1	-1	2
$2u - 3v$	3	2	-3
$-7u + 11v$	?	-7	11

The Lehmer's GCD algorithm based on the above idea is shown in Alg. 3.2 (see [1, pp.345–348]). In Alg. 3.2, it is assumed that each integer is described as b-adic number and each word (single precision) can represent b-adic p-digit number. Moreover, d_u and d_v denote the numbers of digits of u and v, respectively.

Algorithm 3.2 Lehmer's GCD algorithm

Require: u, v $(u \leq v)$
Ensure: d, s, t $(d = \gcd(u, v),\ d = su + tv)$

$(s_0, t_0) \leftarrow (1, 0)$
$(s_1, t_1) \leftarrow (0, 1)$
$i \leftarrow 1$
if $v = 0$ **then**
  $(d, s, t) \leftarrow (u, s_i, t_i)$
  return (d, s, t)
else
  $k \leftarrow \max(d_u - p, 0)$
  $(u', v'') \leftarrow (\lfloor u/b^k \rfloor, \lfloor v/b^k \rfloor)$
  $(u'', v') \leftarrow (u' + 1, v'' + 1)$
end if
if $v'' = 0$ **then**
  $q \leftarrow \lfloor u/v \rfloor$
  $(s_{i+1}, t_{i+1}, r) \leftarrow (s_{i-1}, t_{i-1}, u) - q(s_i, t_i, v)$
  $(u, v) \leftarrow (v, r)$
  $i \leftarrow i + 1$
else
  $(s'_0, t'_0) \leftarrow (1, 0)$
  $(s'_1, t'_1) \leftarrow (0, 1)$
  $j \leftarrow 1$
  $q' \leftarrow \lfloor u'/v' \rfloor$, $q'' \leftarrow \lfloor u''/v'' \rfloor$
  if $q' = q''$ **then**
    $(s'_{j+1}, t'_{j+1}) \leftarrow (s'_{j-1}, t'_{j-1}) - q'(s'_j, t'_j)$, $j \leftarrow j + 1$
    $(s_{i+1}, t_{i+1}) \leftarrow (s_{i-1}, t_{i-1}) - q'(s_i, t_i)$, $i \leftarrow i + 1$
    goto 21:
  end if
  $u \leftarrow s'_{j-1}u + t'_{j-1}v$
  $v \leftarrow s'_j u + t'_j v$
end if
goto 4:

3.3. Fast Chinese remainder algorithm

Moreover, we applied a bottom-up type divide and conquer method to speed up the Chinese remainder algorithm, i.e., solved a system of congruence equations by tournament style CRT (see [2] p.290). We also used the fast EEA to accelerate more.

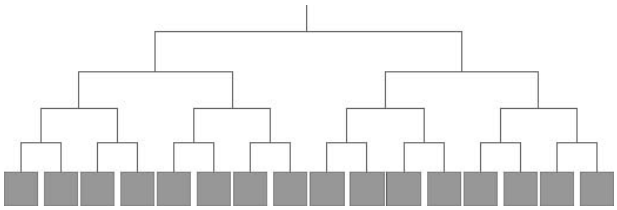

Figure 3. A schematic diagram of the fast Chinese remainder algorithm.

4. Numerical experiments

We implemented the rational number reconstruction method stated above on CPU and GPU, and evaluated it

by some numerical experiments. In this implementation, we used algorithms stated in §3 for more fast calculation. Hereafter, we denote by GMP the calculation using the library GMP[1], which computes rational expressions directly and does not use the reconstruction method. And we denote by CPU and GPU the implementations on CPU and GPU, respectively. Both of them compute rational expressions by using the rational number reconstruction method using Chinese remainder theorem.

We applied those three GMP, CPU and GPU programs to three types of rational calculations. In the experiments, each rational number input as element of vectors/matrices is randomly selected in the range $[-2^{31}, 2^{31}-1]$ for numerator and $[1, 2^{31}-1]$ for denominator. The computation environment is as follows: OS: Vine Linux 6 (64bit), CPU: intel Core i7 4770K, GPU: nVidia GTX 980, RAM: 16GB, GPU library: CUDA 7.5, Compiler: GCC 4.9.3, other libraries: GMP 6.1.0 and Boost 1.60.0.

4.1. Inner product

At first we calculated inner products. That is, we chose

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = x_1 y_1 + \cdots + x_n y_n$$

as the object rational functions and applied Alg. 2.4 to it. In the algorithm, we used 16384 moduli and carried out experiments for $n = 2^7, 2^8, \ldots, 2^{12}$ as the dimension of vectors.

The experimental results are shown in Fig. 4 as a double logarithmic chart. The horizontal axis represents the dimension of vectors, while the vertical one the computation time (sec). Each straight line on the graph is the fitted curve by a least square method.

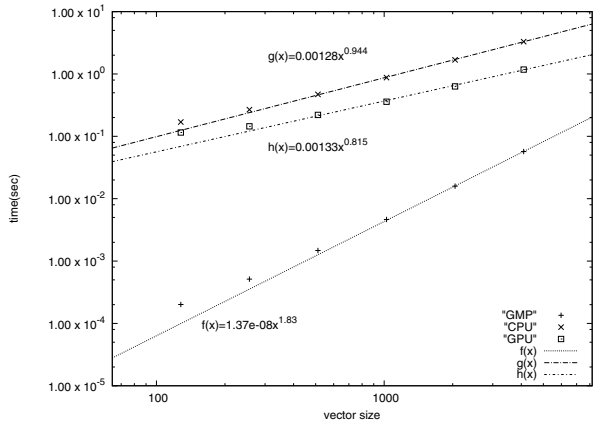

Figure 4. Computation time for inner products.

From Fig. 4, one can see that GMP is much faster (over 10 times) than CPU and GPU. But for the incline of the fitting curve, GPU is the smallest one among them. In this experiments, because of the restriction of GPU memory, we could not compute for vectors with higher dimension. But we expect that GPU would be faster than GMP for a vector with sufficiently high dimension.

1. The GNU Multiple Precision Arithmetic Library.

4.2. Frobenius normal form

It is reported that a modular algorithm accelerates a converting to Frobenius normal form [4]. So we applied our implementation of Alg. 2.4 to this problem. In the algorithm, we used 1024 moduli and carried out experiments for matrices of order $n = 10, 20, \ldots, 140$.

The experimental results are shown in Fig. 5 as a double logarithmic chart, where the horizontal axis represents the order of matrices, while the vertical one the computation time (sec). Each straight line on the graph is the fitted curve by a least square method.

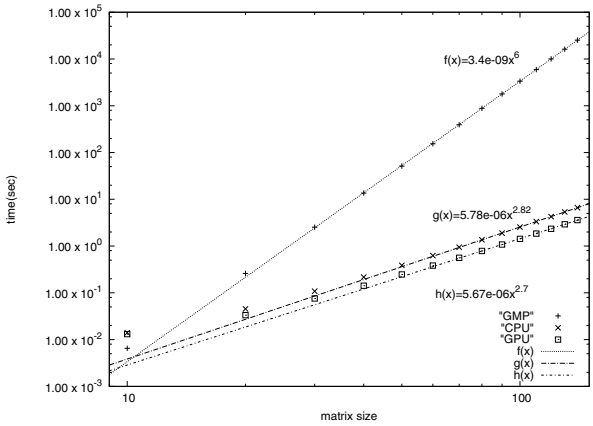

Figure 5. Computation time for Frobenius normal form.

From Fig. 5, one can see that GPU is the fastest for the order $n > 10$. Furthermore, from the result of fitting by a least square method, one can see that the order of computation time of GMP (using no rational number reconstruction) is $O(n^6)$, while that of GPU (using rational number reconstruction) is $O(n^{2.7})$, it means that the GPU (or CPU) is essentially faster than GMP.

Moreover, for this problem, we observed the computation times varying the number of moduli (see Tab. 4). In this experiment, we fixed the order of matrices $n = 250$. "MOD" in Tab. 4 denotes the process converting input rational numbers to their modular forms, "CALC" the process converting to Frobenius normal form under modular arithmetic, "CRT" the process of Chinese remainder algorithm and "RR" the process reconstructing rational numbers from their modular forms.

Comparing CALC rows of CPU and GPU, one can notice that computation time of CPU is increasing almost proportional to the number of moduli, while that of GPU is increasing slightly but not proportionally until the number of moduli exceeds 8192. It is explained by the fact that the more threads per core, the more efficiently the computation is performed, in general.

Anyway, in the case of the largest number of moduli (16384), GPU is about 13.8 times faster than CPU in the process CALC and about 8.3 times faster in total.

TABLE 4. COMPUTATION TIME FOR VARIOUS NUMBERS OF MODULI.

Number of moduli		1024	2048	4096	8192	16384
CPU	MOD	1.488949	2.988731	5.937348	11.884419	23.839760
	CALC	36.717916	73.977371	148.956431	298.986495	598.443245
	CRT	0.263690	0.704150	1.639970	4.013283	10.416749
	RR	0.028354	0.045215	0.077809	0.150679	0.526969
	TOTAL	38.498928	77.715487	156.611581	315.034898	633.226744
GPU	MOD	1.471862	2.974136	5.947215	11.925243	23.816390
	CALC	17.269066	18.130380	19.278103	21.678577	43.513525
	CRT	0.240045	0.561186	1.325094	3.216123	8.063184
	RR	0.018667	0.026355	0.043883	0.081890	0.421737
	TOTAL	19.010735	21.711882	26.630083	36.971599	75.951404

4.3. Determinant

Finally, we applied Alg. 2.4 to the calculation of determinant, because it is widely used and expected to be accelerated by the rational number reconstruction method. In order to calculate determinants, we used the LU decomposition algorithm and applied our implementation to it with 1024 moduli. We did the experiment on the matrices of order $n = 10, 20, \ldots, 400$.

The experimental results are shown in Fig. 6 as a double logarithmic chart, where the horizontal axis represents the order of matrices, while the vertical one the computation time (sec). Each straight line on the graph is the fitted curve by a least square method.

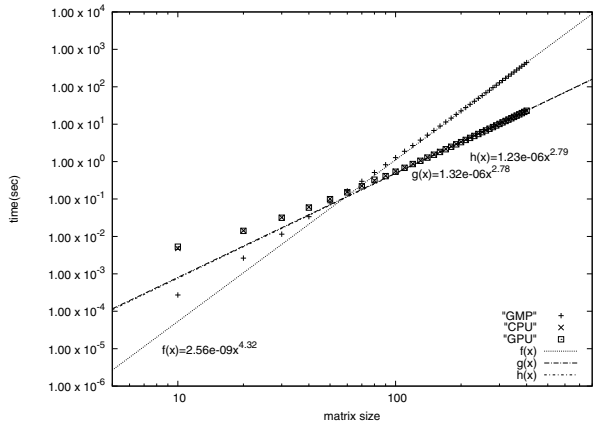

Figure 6. Determinant's calculation time (sec).

From Fig. 6, one can see that GPU is the fastest for the order $n > 50$. Furthermore, from the result of fitting by a least square method, one can see that the order of computation time of GMP (using no rational number reconstruction) is $O(n^{4.3})$, while that of GPU (using rational number reconstruction) is $O(n^{2.8})$, it means that the GPU (or CPU) is also essentially faster than GMP in this calculation.

5. Discussion

From the viewpoint of computational order (measured from the numerical experiments) with respect to the size of problem (i.e. the dimensions of vectors or the degrees of matrices), our implementations CPU and GPU (with rational number reconstruction) were faster than GMP (without it) in the numerical experiments mentioned in §4. Actually, the calculation times of CPU and GPU for Frobenius normal forms or determinants were shorter than that of GMP. But the calculation times of CPU and GPU for inner products were not shorter than that of GMP in the range of experiments. We guess the difference arose because the intermediate expression swells did not occur in the computations of inner products, while they did occur in the computations of Frobenius normal form and determinant of matrices. For another reason, we note that the computation complexity per number of rational elements of inner product, Frobenius normal form and determinant of matrix are $O(1)$, $O(n \log n)$ and $O(n)$, respectively.

Regarding the difference between CPU and GPU, we observed that GPU became more accelerated than CPU as increasing the number of moduli. One of our experiments showed that GPU is about 8.3 times faster than CPU in total and about 13.8 times in main part of computation.

6. Conclusion

By the three types of numerical experiments, we got some partial knowledge about when the rational reconstruction method using CRT accelerates computations. Moreover, we showed that GPU is faster than and has advantage to CPU in some cases, by numerical experiments.

But there is a problem that the size of input data and the number of moduli used in the algorithm are restricted because GPU has not so much memories. One may overcome this problem by dividing a computation into some smaller parts and applying the algorithm to each of them. Related to this problem, when it is difficult to estimate the size of rational numbers of final result, one must use rather too much moduli and consumes much memory, in a result.

For future work, we are planning to implement more operations, to accelerate the computation more and to resolve the problems stated above.

References

[1] Donald E. Knuth, The Art of Computer Programing **2** Seminumerical Algorithms, Addison Wesley, 1998.

[2] Joachim von zur Gathen, Jürgen Gerhard, Modern Computer Algebra, Cambridge University Press, 1999.

[3] Takehiro Tanaka, Hirokazu Murao, An Efficient Method for Multiple-Precision Integer Arithmetics Using GPU, IPSJ SIG Technical Report **2010-HPC-124**, No.2. (In Japanese)

[4] Shuichi Moritsugu, A Practical Implementation of Modular Algorithms for Frobenius Normal Forms of Rational Matrices, IPSJ journal **45**, No.6.

[5] T. Kan, M. Iri, Jordan Normal Forms, University of Tokyo Press, 1982. (In Japanese)

[6] Peter Montgomery, Modular Multiplication Without Trial Division, Math. Computation, **44**, 519–521, 1985.

[7] CUDA Toolkit Documentation v7.5, https://docs.nvidia.com/cuda/.

[8] Hikaru Samukawa, A Proposal of Rational BLAS for Numerical Linear Algebra with Rational Number Arithmetic, HPCS**2014** 57–64. (In Japanese)