Determinant Computation on the GPU using the Condensation Method

Sardar Anisul Haque[1], **Marc Moreno Maza**[2]
University of Western Ontario, London N6A 1M8, Canada
E-mail: [1] shaque4@csd.uwo.ca, [2] moreno@csd.uwo.ca

Abstract. We report on a GPU implementation of the condensation method designed by Abdelmalek Salem and Kouachi Said for computing the determinant of a matrix. We consider two types of coefficients: modular integers and floating point numbers. We evaluate the performance of our code by measuring its effective bandwidth and argue that it is numerical stable in the floating point number case. In addition, we compare our code with serial implementation of determinant computation from well-known mathematical packages. Our results suggest that a GPU implementation of the condensation method has a large potential for improving those packages in terms of running time and numerical stability.

1. Introduction

The celebrated algorithm of Charles Lutwidge Dodgson [1] (also known as Lewis Carroll) for computing the determinant of a square matrix $A = (a_{i,j} \mid 0 \leq i, j \leq n-1)$ of order n is a popular trick among students. It is, indeed, much easier to perform by hand on paper than the other classical methods, such as those based on minor expansion or Gaussian elimination. This is due to its amazing data traversal pattern. Each transformation step, from one array to the next one, is a streaming process, called a *condensation*. Dodgson's Algorithm can be executed as a stencil computation: the input data array is transformed into its determinant through $n-1$ successive data arrays. This method suffers, however, from a serious algebraic limitation: it may fail to compute the targeted determinant. Indeed, after each condensation, the newly generated matrix should have no zero elements in its interior [1] for the next condensation step to take place. The interior of A is the submatrix $\text{int}(A) = (a_{i,j} \mid 0 < i, j < n-1)$. One can sometimes reduce to this case by combining rows or columns. When this is not possible, the algorithm terminates without producing any answers. In [2], Abdelmalek Salem and Kouachi Said have solved this difficulty by introducing another type of condensation.

One can easily realize that the condensation method (Dodgson's original method and the improved one by Salem and Said) can be executed in parallel. Moreover, we argue in this paper that its data traversal pattern makes it a good candidate for an implementation within a concurrency platform based on data-parallelism such as CUDA [5].

We report on an implementation of the algorithm described in [2] on GPU using CUDA. We consider two types of coefficients: modular integers and floating point numbers. In the first case, our contribution is to show that the condensation method can be implemented efficiently in terms of memory bandwidth, leading to a very competitive code with respect to popular software packages for computing determinant over finite fields (i.e. with modular integer coefficients).

In the floating point case, our contribution is to show that the condensation method can be implemented efficiently in terms of numerical stability. We observe that the condensation method computes, in some sense, a factorization of the determinant. To take advantage of this fact, we use a new algorithm to compute the product of those factors such that, if overflow/underflow can be avoided then computations are ordered in a way that overflow/underflow is indeed avoided. The challenge is to keep the intermediate products within the range of machine floats; our solution is described in Section 4.

The organization of the paper is as follows. We describe the condensation method in Section 2. Its GPU implementation is presented in Section 3 and 4 for the finite field and floating point case respectively. Both of these two sections contain the corresponding experimental results. Concluding remarks are in Section 5.

2. The condensation method

In this section, we first review the condensation method described in [2]. We will then analyze the algebraic complexity and cache complexity of this condensation method.

2.1. The formula of Salem and Said

As mentioned in the introduction, the authors of [2] have solved the algebraic limitation of Dodgson's Algorithm by introducing another type of condensation, which we summarize below. The input is a square matrix A of order $n > 2$. If the first row of $A = (a_{i,j} \mid 0 \leq i, j \leq n-1)$ is the null vector then the determinant of A is zero and the process terminates. Otherwise, let ℓ be the smallest column index of a non-zero element in the first row of A. The *condensation step* produces a matrix $B = (b_{i,j})$ of order $n-1$ defined by:

$$b_{i,j} = \begin{vmatrix} a_{0,\ell} & a_{0,j+1} \\ a_{i+1,\ell} & a_{i+1,j+1} \end{vmatrix}$$

for $j \geq \ell$ and by $b_{i,j} = -a_{i+1,j}a_{0,\ell}$ for $j < \ell$. The key relation between A and B is the following:

$$\det(A) = \det(B)/(a_{0,\ell})^{n-2} \tag{1}$$

We call $a_{0,\ell}$ the *pivot* of the condensation step. Formula (1) implies that the condensation method of Salem and Said computes $\det(A)$ as a product of powers of the inverse of the successive pivots. In the finite field case, this product can be accumulated in a variable, that is, this variable is updated after each condensation step. In the floating point number case, these powers can be accumulated in a list so that their product can be performed in a way that overflow/underflow is avoided, if possible, as we shall see in Section 4.2.

2.2. The algebraic complexity of the condensation method

Algebraic complexity estimates are given for the RAM model with memory holding a finite number of s-bit words, for a fixed s [9]. Each condensation step involves two matrices: A and B of order n and $n-1$, respectively. Computing B from A requires $2(n-1)^2$ multiplications and $(n-1)^2$ subtractions considering that ℓ refers to the first column (which is the worst case for computing B). The best case happens when ℓ is the last column. When this happens each condensation requires $(n-1)^2$ multiplications. The number of operations involved in finding ℓ is linear in n. The whole algorithm takes at most $n-2$ condensation steps before terminating. So, the total cost for computing the determinant is bounded by $O(n^3)$ arithmetic operations. Moreover, in the worst case, a precise account is $n^3 - 3/2n^2 + 1/2n - 3$, which is comparable to the worst case of Gaussian Elimination.

2.3. The cache complexity of condensation method

Cache complexity estimates are given for the ideal cache model. The ideal cache model [10] is a fully associative cache. Its cache replacement policy is optimal in that the cache line to be evicted is one which will be required again furthest in future.

Before estimating the cache complexity of a condensation step, we need to describe the data structures used to represent a square matrix of order n in our implementation. We represent A by a one-dimensional array $\alpha[0, 1, \ldots, n^2 - 1]$ of size n^2. We use *column major layout.* that is, the sub-array $\alpha[i, i + 1, \ldots, i + n - 1]$, for $i = 0, n, 2n, \ldots (n - 1)n$ represents the i-th column. In particular the element $a_{i,j}$ is stored in $\alpha[i + j * n]$.

Consider an ideal cache of Z words, with cache line of L words. To make the analysis simple, we assume that n is large enough such that one column of A does not fit into the cache. Let α and β be two one-dimensional arrays of size n^2 and $(n-1)^2$, representing the input matrix A and output matrix B of the condensation method, respectively. In each condensation step, matrix B is created from A. Assume that, our algorithm will compute B sequentially column-wise. This involves the following data traversals:

- Each element of B is visited only once.
- Each element of A (if it is neither in first row nor in ℓ-th column) is visited only once.
- The ℓ-th column of A is scanned $n - 1$ times.
- Each element of the first row of A is visited $n - 1$ times consecutively.

It follows that one condensation step incurs $2(n - 1)^2/L + n/L + 3$ for α and $(n - 1)^2/L + 1$ for β, thus $3(n - 1)^2/L + n/L + 4$ cache misses in total. Summing over the condensation steps for $k = Z + 1 \cdots n$ (that is, those for which one column does not fit it in cache) we obtain

$$\frac{(n - Z)\left(n^2 - n + Z^2 - Z + Zn + 1 + 4\,L\right)}{L} \tag{2}$$

Therefore, asymptotically, the ratio between the algebraic complexity and the cache complexity is L. This is similar to Gaussian Elimination. However, the condensation method works in a more data-oblivious way: at each condensation step, apart from the search of the pivot, the same operations are performed independently of the data. This regularity pattern facilitates scheduling, in particular hardware scheduling as it is the case on a GPU. Gaussian Elimination does not have this feature. Indeed, permutations of rows and columns may be needed before proceeding to the next step.

3. GPU implementation: the finite field case

As mentioned in the introduction, the GPU implementation reported in this paper handles two types of coefficients, namely modular integers and floating point numbers. In both cases, each coefficient is stored in a fixed number of machine words and hardware arithmetic is used as much as possible for efficiency consideration. This latter property is more difficult to achieve in the case of modular integers and we discuss it in Section 3.2 Numerical stability is the challenge of floating point number arithmetic and we address it in Sections 4.1 and 4.2. Other types of coefficients, such as multi-precision integers, could be also considered and we leave it for future work.

Before describing the issues specific to the finite field case (in other words to modular integers) we present the part of our implementation which is common to both scenarios. More precisely, we discuss in Section 3.1 our GPU implementation strategy, in particular the mapping between thread blocks and data. In section 3.3, we report on our experimentation with the condensation method for matrices over finite fields. This, of course, is primarily dedicated to evaluate the performance of our GPU implementation, but also to compare it with serial implementations

of determinant computation available in computer algebra packages One of our goals is to understand to which extent GPU implementation could improve those packages.

3.1. Data mapping

Each condensation step is performed by one CUDA kernel call. The matrices α and β, introduced in Section 2.3, are stored in the global memory of GPU. After each condensation step, instead of copying β back to CPU main memory, we simply "swap the pointers" to these arrays.

In practice, we find that the index ℓ is small. So we dedicate one kernel call, with a single thread in a single block, to find ℓ. Once we get ℓ, we compute the $(n-2)$-th power of the inverse of $\alpha[\ell * n]$. We will call it *pivot*. We also store the product of the successive pivots in this kernel call.

The kernel performing a condensation step uses one-dimensional blocks and threads. Let T be the number of threads in a block. Each thread is responsible to compute t elements of the array β (representing B). So the total number of blocks required to compute β is $\lceil (n-1)^2/(Tt) \rceil$. Consider thread i is in block j. Then this thread is responsible for computing $\beta[Ttj + it, Ttj + it + 1, \ldots Ttj + it + t - 1]$.

3.2. Finite field arithmetic

Multiplying two elements a, b modulo a prime number p is obviously a key routine. Unlike the case of single and double precision floating point arithmetic, the operation $(a, b, p) \longmapsto (ab) \bmod p$, for $a, b, p \in \mathbb{Z}$, is not provided directly by the hardware. This operation is thus an efficiency-critical low-level software routine that the programmer must supply. When p is a machine word size prime, which is the assumption in this paper, two techniques are popular in the symbolic computation community.

The first one takes advantage of hardware floating point arithmetic. We call `double_mul_mod` our implementation of this technique, for which our CUDA code is shown below. The fourth argument `pinv` is the inverse of `p` which is precomputed in floating point.

```
__device__ int double_mul_mod(int a, int b, int p, double pinv) {
   int q  = (int) ((((double) a) * ((double) b)) * pinv);
   int res = a * b - q * p;
   return (res < 0) ? (-res) : res;
}
```

In our implementation, double precision floating point numbers are encoded on 64 bits and make this technique work correctly for primes p up to 30 bits.

The second technique, called the *Montgomery reduction* relies only on hardware integer arithmetic. We refer to Montgomery's paper [11] for details. We have experimented both approaches in [12]. Our CUDA implementation favors the `double_mul_mod` trick.

3.3. Experimental results

We generate random integer matrices modulo a prime number p of machine word size. The order of our test matrices varies from 10 to 4000. We conduct all our experiments on a GPU NVIDIA Tesla 2050 C.

We use *effective memory bandwidth* to evaluate our GPU code. The effective memory bandwidth (measured in GB/seconds) of a kernel run is, by definition,

- the amount of data traversed in the global memory of the GPU during the kernel run,
- divided by the running time of the kernel.

Following a principle proposed by Greg Ruetsch and Paulius Micikevicius in [7], we compared the effective memory bandwidth of our kernel to that of a *copy kernel*, that is, a kernel that

simply performs one memory copy from one place to another place in the global area of GPU. Such benchmark kernel can be regarded as a good practical measure of the maximum memory bandwidth of a kernel.

For matrix of order of 3000, the effective memory bandwidth of the copy kernel and our condensation method (with modular integer coefficients) on our card are 96 GB/s and 18.5 GB/s respectively.

Our effective memory bandwidth results show that our code is reasonably efficient considering the following two facts:

- the index calculation in our code is not straightforward and
- finite field arithmetic (see Section 3.2) is time consuming.

Figure 1 reports on the memory bandwidth of our CUDA code for different matrix orders.

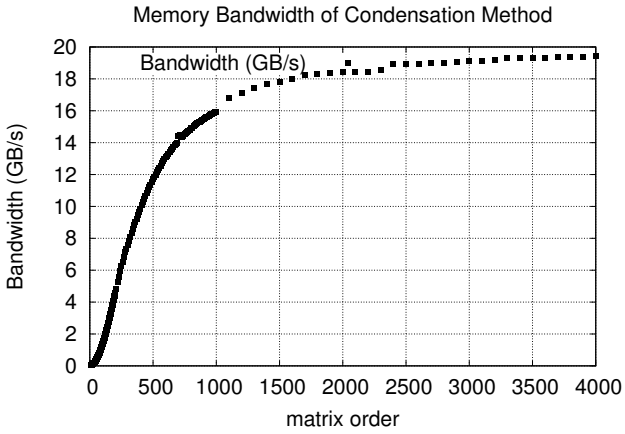

Figure 1. Effective memory bandwidth of condensation method.

To conclude this section, we compare our CUDA code for computing determinants over finite fields with its counterpart in two popular mathematical software packages, namely MAPLE [4] and NTL [3].

Figure 2 compares the computing time between our CUDA code for the condensation method and the NTL determinant code, both with modular integer coefficients. It shows clearly that the condensation method in CUDA outperforms NTL code for computing determinant over finite fields. For example, when $n = 4000$, the condensation method in CUDA takes about 9 seconds, while NTL code takes about 850 seconds.

Figure 3 compares the computing time between our CUDA code for the condensation method and MAPLE's determinant command over finite fields. It shows clearly that the condensation method in CUDA outperforms MAPLE code for computing determinant over finite fields. For example, when $n = 4000$, the condensation method in CUDA takes about 9 seconds, whereas MAPLE code takes about 90 seconds.

4. GPU implementation: the floating point case

In this section, we consider the case of matrices with floating point number coefficients. We adapt our CUDA code described in Section 3 to this new scenario. The modifications are described in Section 4.1. One potential challenge that we found is to multiply the successive pivots. Mathematically, the problem is to multiply a sequence of floating values where the intermediate results might not be in the range of the floating point number data type while the final results might be. We will state the problem and our solution in Section 4.2. We conclude the section by providing experimental results.

4.1. Finding the pivots

Instead of taking the first nonzero from the left in the first row of A, we choose the nonzero element of the first row that is closest to value 1.0; let use call $p = a_{0,\ell}$ this element. We have verified that this modification of the original algorithm in [2] does not invalidate the expected result, namely the determinant. For simplicity, we are describe the procedure for matrices A and B instead of the arrays α and β. Once p is chosen, all elements in the ℓ-th column are divided by p including $a_{0,\ell}$. Thus we modify Formula (1) as follows:

$$\det(A) = \det(B) * p$$

We will call p the pivot for this floating point number implementation.

The benefits of the above transformation are as follows.

- $a_{0,\ell}$ becomes 1.0. So we need neither computing the $(n-2)$-th power of it nor performing any division at the end.
- By choosing an element that is the closest to 1.0, we are expecting to reduce the potential of overflow/underflow.

4.2. Multiplication of the successive pivots

We first state the problem that we wish to address. Consider an array $x[0, 1, \ldots, k-1]$ of k floating point numbers, encoded by a floating point number data type of fixed precision. Then the problem is to write an algorithm for computing the product $X = \prod_{i=0}^{k-1} x[i]$, assuming that X fits within the range of the given floating point number data type. Our solution to this problem is stated as Algorithm 1 hereafter.

We give a sketch of the proof of Algorithm 1. We observe that multiplications occur at Lines 6, 17 and 25. The multiplication at Line 6 cannot lead to overflow/underflow since $|q1| \leq 1 \leq |q2|$ holds. If Lines 17 or 25 would lead to overflow/underflow, this would bring a contradiction to our hypothesis.

We estimate the running time of Algorithm 1. The first *while* loop runs in linear time with the number of elements in x. The second *while* loop runs $m-1$ times considering there exists m elements in R. Each of the iteration takes $O(m)$ time. So the time complexity of the second *while* loop is $O(m^2)$. Considering the inequality $k \geq m$, the time complexity of the Algorithm is $O(k^2)$.

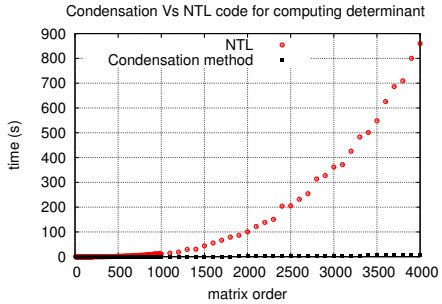

Figure 2. CUDA code for condensation method and determinant on NTL over finite field.

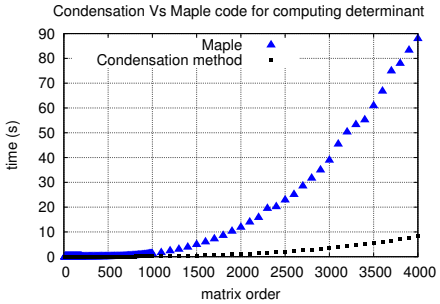

Figure 3. CUDA code for condensation method and determinant on MAPLE over finite field.

Algorithm 1 MulSuccPivot (x)

Create a stack $S1$ of elements of x in $[-1.0, 1.0]$.
Create a stack $S2$ of the other elements of x not in $S1$.
while both $S1$ and $S2$ are nonempty **do**
    $q1 = pop(S1)$
    $q2 = pop(S2)$
    $q = q1 * q2$
    if q is in $[-1.0, 1.0]$ **then**
        $push(q, S1)$
    else
        $push(q, S2)$
    end if
end while
if stack $S1$ is not empty **then**
    make a list R with the elements in $S1$
    while R has more than one element **do**
        select $r1$ and $r2$ in R such that $r1$ and $r2$ are closest to 0 and $|1.0|$ respectively.
        $r = r1 * r2$
        delete $r1$ and $r2$ from R
        insert r into R
    end while
else
    make a list R with the elements in $S2$
    while R has more than one element **do**
        select any $r1$ and $r2$ in R.
        $r = r1 * r2$
        delete $r1$ and $r2$ from R
        insert r into R
    end while
end if
return R

4.3. Experimentation

For the experimentation in the case of floating point coefficients matrices with MAPLE, we use the *Determinant* command of the *LinearAlgebra* package. In this case, MAPLE may not have the best possible implementation, since MAPLE's primary purpose is symbolic computation. However, MATLAB has certainly a competitive implementation for floating point coefficients. Indeed, efficiently supporting numerical linear algebra is the primary goal for this cutting-edge software.

For small finite field coefficients, the best serial algorithm is simply Gaussian elimination, which is what MAPLE is using. Therefore, in the case of modular integers, our comparison reported in Section 3.3 is also meaningful.

In order to investigate the numerical stability of our GPU implementation of the condensation method, we use the infamous Hilbert matrix $H_{ij} = \frac{1}{i+j-1}$, which is a canonical example of ill-conditioned matrix. This matrix is non-singular, for each value of n. However, for n large enough, any determinant computation of this matrix using a fixed precision floating point number arithmetic will return zero.

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{bmatrix}$$

In the tables below, we compare determinant computation of the Hilbert matrix with

- MAPLE using multi-precision floating point number arithmetic (thus software floating point number),
- MATLAB using double-precision floating point number arithmetic,
- our CUDA implementation using double-precision floating point number arithmetic.

We observe that:

(i) despite of the use of multi-precision floating point, MAPLE is less accurate than MATLAB and our CUDA implementation (this was checked by computing the exact value of the determinant using rational number arithmetic),

(ii) our CUDA implementation of the condensation method can compute determinants of much larger order than MATLAB,

(iii) our CUDA implementation is also competitive with MATLAB in terms of running time.

5. Conclusion

MAPLE and MATLAB commands for computing matrix determinants combine many different state-of-the-art algorithms. On a given input, MAPLE and MATLAB determinant commands choose one of these algorithms by considering the types of the coefficients and the combinatorial properties (size, sparsity) of the input matrix. These choices are heavily tuned since linear algebra is, in the case of MAPLE, at the core of its symbolic routines while it is, in the case of MATLAB, at the core of the whole system. Therefore, comparing our code against those systems is meaningful. From our experimental results, it is clear that the condensation method implemented on the GPU is a promising candidate for computing determinants of matrices with both modular integer coefficients and floating point number coefficients.

Though it seems unfair in the first place that we compare our parallel code with serial codes in mathematical packages, our primary objective is to propose parallel algorithms for computing determinants within mathematical software packages, such as MAPLE and MATLAB. Actually, these two systems are already able today to take advantage of multicore processors and GPUs for certain types of computations. Therefore our objectives are meaningful and motivated by our active cooperation with the Maplesoft company developing MAPLE. We believe that a GPU implementation of the condensation method can be used to improve the efficiency, in terms of running time and numerical stability, of existing mathematical software packages.

Acknowledgments

We like to thank Dr. Wei Pan (Intel Corporation) for discussions around our CUDA implementation in the finite field case. We also like to thank Dr. Jürgen Gerhard (Maplesoft) for his suggestions in the floating point number case.

Matrix order	MAPLE	MATLAB	Condensation Method on GPU
5	0.3239712e-11	3.749295e-12	3.74967e-12
6	-0.1037653175e-16	5.367300e-18	5.36556e-18
7	-0.2940657217e-22	4.835803e-25	4.44292e-25
8	-0.2156380381e-28	2.737050e-33	-3.92813e-33
9	-0.1692148341e-35	9.720265e-43	-2.79235e-41
10	0.4704819751e-42	2.164405e-53	-4.44342e-50
15	0.1386122551e-74	-2.190300e-120	-9.47742e-103
20	0.4711757502e-106	-1.100433e-195	3.81829e-164
25	-0.4092672466-139	5.482309e-274	-3.82134e-239
30	-0.2087134536-174	0	-2.50914e-319
35	0.6863051439e-205	-	3.50293e-398
40	0.3354475665e-237	-	-7.42227e-479
70	-0.1605231989e-443	-	-1.42973e-961
100	-0.1344119185e-667	-	1.96009e-1467
200	-0.1635472167e-1423	-	9.43651e-3169
295	-0.1313897019e-2117	-	3.27673e-4811
300	0.4832058492e-2154	-	-1.95564e-4897
320	0.1012376674e-2298	-	7.2904e-4951
340	0.3198288621e-2442	-	-8.67557e-4949*2.08848e-644
360	0.6712616355e-2593	-	9.84118e-4938*8.32678e-1006
380	-0.1532669346e2736	-	-3.28068e-4950*-6.51644e-1341
400	-0.4230797452e-2881	-	-6.19676e-4945*6.56337e-1696
500	-0.1956609252e-3608	-	1.40177e-4939*-2.22223e-3444
600	-0.4139972675e-4335	-	-2.55164e-4950*1.99856e-4945*6.19736e-232
800	0.4570493645e -5853	-	4.25009e-4940*-2.21715e-4940*-2.17891e-3739

Table 1. Determinant of Hilbert Matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.

References

[1] C. L. Dodgson. Condensation of Determinants, Proceedings of the Royal Society of London, 15(1866), 150-155.

[2] Abdelmalek Salem, and Kouachi Said. Condensation of Determinants, `http://arxiv.org/abs/0712.0822`.

[3] NTL: A library for doing number theory. `http://www.shoup.net/ntl`.

[4] Maple: The essential tool for mathematics and modeling. `http://www.maplesoft.com`.

[5] NVIDIA developer zone. `http://developer.nvidia.com`.

[6] Matlab- The language of technical computing. `http://www.mathworks.com`.

[7] G. Ruetsch and P. Micikevicius. Optimizing MatrixTranspose in CUDA, NVIDIA Corporation, 2009.

[8] W. Pan. Algorithmic Contributions to the Theory ofRegular Chains, PhD Thesis, The University of Western Ontario, Canada, 2011.

[9] J. E. Savage. Models of Computation, Addison-Wesley Longman, Boston, MA, USA, 1998.

[10] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran. Cache-Oblivious algorithms. FOCS'99: Proc. of the 40th Annual Symp. on Foundations of Comp. Sc., 1999

[11] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985

[12] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *Proc. ISSAC'09*, pages 239–246, New York, NY, USA, 2009. ACM Press.

Matrix order	MAPLE	MATLAB	Condensation Method on GPU
5	0.004	0	0.000530
6	0.008	0	0.000570
7	0.012	0	0.000595
8	0.008	0	0.000631
9	0.012	0	0.000741
10	0.012	0	0.000447
15	0.016	0	0.000964
20	0.016	0	0.001078
25	0.020	0	0.001271
30	0.024	-	0.001460
35	0.044	-	0.001671
40	0.036	-	0.001896
70	0.188	-	0.003083
100	0.588	-	0.005145
200	5.988	-	0.012488
295	20.733	-	0.023402
300	21.661	-	0.023759
320	26.741	-	0.026633
340	31.677	-	0.029433
360	38.150	-	0.032401
380	46.146	-	0.035940
400	54.099	-	0.038955
500	104.334	-	0.058193
600	187.151	-	0.081969
800	467.541	-	0.147037

Table 2. Time(s) Required to compute determinant of Hilbert Matrix by MAPLE, MATLAB, and condensation method on both CPU and GPU.