Plain Polynomial Arithmetic on GPU

Sardar Anisul Haque, Marc Moreno Maza

University of Western Ontario, Canada

E-mail: `shaque4@uwo.ca, moreno@csd.uwo.ca`

Abstract. As for serial code on CPUs, parallel code on GPUs for dense polynomial arithmetic relies on a combination of asymptotically fast and plain algorithms. Those are employed for data of large and small size, respectively. Parallelizing both types of algorithms is required in order to achieve peak performances. In this paper, we show that the plain dense polynomial multiplication can be efficiently parallelized on GPUs. Remarkably, it outperforms (highly optimized) FFT-based multiplication up to degree 2^{12} while on CPU the same threshold is usually at 2^6. We also report on a GPU implementation of the Euclidean Algorithm which is both work-efficient and runs in linear time for input polynomials up to degree 2^{18}, thus showing the performance of the GCD algorithm based on systolic arrays.

1. Introduction

Until the advent of multicore architectures, algorithms subject to effective implementation on personal computers were often designed with *algebraic complexity* as the main complexity measure and with sequential running time as the main performance counter [15, 16, 17, 4, 10]. Nevertheless, during the past 40 years, the increasing gap between memory access time and CPU cycle time, in favor of the latter, brought another important and practical efficiency measure: *cache complexity* [14, 8]. In addition, with parallel processing becoming available on every desktop or laptop, the *work* and *span* of an algorithm expressed in the fork-join multithreaded model [9, 4] have become the natural quantities to compute in order to estimate *parallelism.*

These complexity measures (algebraic complexity, cache complexity, parallelism) are defined for computation models that largely simplify reality. On multicore architectures, several phenomena (memory wall, true/false sharing, scheduling costs, etc.) limit the performances of applications which, theoretically, have a lot of opportunities for concurrent execution. One infamous example are *Fast Fourier Transforms (FFTs).* For this type of calculation, not only the memory access pattern, but also the for-loop parallelization overheads, restrict linear speedup to input vectors of very large sizes, say 2^{20}, according to [21, 22]. In contrast, serial FFT code provide high-performance even for input vectors of relatively small sizes, say 2^{10}. This is the case with the standard libraries FFTW [7], NTL [25] and Spiral [24]. As a consequence, higher level algorithms, that heavily rely on FFTs in their serial implementation, require additional supporting routines for small/average size problems, when targeting implementation on multicore architectures. Examples of such higher level algorithms are fast evaluation and fast interpolation based on sub-product tree techniques, see Chapter 10 in [10].

Graphics processing units (GPUs) offer a higher level of concurrent memory access than multicore architectures. Moreover, thread scheduling is done by the hardware, which reduces for-loop parallelization overheads significantly. Despite of these attractive features, and as reported in [19], highly optimized FFT implementation on GPUs are not sufficient to support the parallelization of higher level algorithms, such as dense univariate polynomial arithmetic. To give a figure, for vectors of size 2^{18} and 2^{26}, speedup factors (w.r.t. a C serial implementation) are 9 and 37 respectively on a NVIDIA Geforce GTX 285 running CUDA 2.2, as reported in [19].

Consider now the fundamental application of polynomial arithmetic: *solving systems of non-linear equations*. Many polynomial systems encountered in practice have finitely many solutions. Moreover, those systems that can be solved symbolically by computer algebra software, such as Maple or Mathematica, have rarely more than 10,000 solutions, see for instance [6]. For this reason, the degrees of univariate polynomials that arise in practice rarely exceed 10,000.

It follows from the above discussion that implementation a polynomial system solver on multicores or GPUs require efficient parallel polynomial arithmetic in relative low degrees, that is, within degree ranges where FFT-based methods do not apply. The study conducted in [3] show that univariate arithmetic based on parallel versions of the Algorithm of Karatsuba and its variants are not effective either in the desired degree ranges. This leads us to consider quadratic (or plain) algorithms for dense univariate multiplication, division and GCD computation. That is, algorithms which run within $O(d^2)$ coefficient operations for polynomials of degree less than d, meanwhile FFT-based algorithms amount to $O(d\log(d)\log(\log(d)))$ coefficient operations, see the landmark textbook [10] for details.

In the present paper, we show that a GPU implementation (with CUDA) of the *plain* univariate multiplication can outperform for fairly large degrees an optimized GPU implementation (with CUDA also) of an FFT-based univariate multiplication. We also report on a GPU implementation of the *plain* univariate division. Section 3 and 4 are dedicated to these multiplication and division CUDA codes, respectively: both contain implementation details, theoretical analysis and experimental results.

We focus on dense polynomial arithmetic over finite fields, since the so-called *modular methods*, such as that presented in [6], allows us to reduce the solving of polynomial systems with rational number coefficients to the solving of polynomial systems over finite fields. To this end, we have realized a preliminary implementation GPU-supported solver for polynomial systems over finite fields. In [19], we report experimental results showing speedup factors of $7, 5$ (w.r.t. a highly optimized serial implementation in C) for bivariate systems with $14, 400$ solutions. This limited speedup is due to the fact that one essential operation of this solver is performed on the host (and thus is not parallelized yet): univariate polynomial GCD computation.

For this reason, in a third part of the present paper, we turn our attention to one of the most challenging parallelization problem in polynomial arithmetic: the *Euclidean Algorithm*. Indeed, there is no parallel version of this algorithm which would be both sublinear and work-efficient[1]. The best parallel version of the Euclidean Algorithm which is work-efficient, is that for systolic arrays, a model of computation formalized by H. T. Kung and C. E. Leiserson [18], for which the span is linear [2]. Multiprocessors based on systolic arrays are not so common (as they are quite specialized to certain operations and difficult to build). However, the recent development of hardware acceleration technologies (GPUs, field-programmable gate arrays, etc.) has revitalized researchers interest in systolic algorithms and more generally in optimizing the use of computer resources for low-level algorithms [1, 11, 13].

We report on a GPU implementation of the Euclidean Algorithm which is work-efficient and which runs in linear time for input polynomials up to degree 2^{18}. As mentioned above, such sizes are sufficient for many applications. Moreover, our GPU code outperforms algorithms for polynomial GCD computations that are asymptotically faster (such as the Half-GCD algorithm, see [26]) but available only as serial CPU code, due to the same parallelization challenges as the Euclidean Algorithm.

[1] Here work-efficient refers to a parallel algorithm in the PRAM model for which the maximum number of processors in use times the span is in the same order as the work of the best serial counterpart algorithm.

2. A manycore machine model

We describe the computation model that we use for analyzing the algorithms that we implement on GPUs. We follow the CUDA execution model with a few natural simplifications. First, we assume that the number of streaming multiprocessors is unlimited. Moreover, we assume that the global memory is unlimited and support concurrent reads and concurrent writes. However, the shared memory and the number of registers of each streaming multiprocessor remain finite and both regarded as small. In addition, moving data between streaming multiprocessors and the global memory remains a penalty, similarly to what happens on a GPU.

On this ideal machine, programs are similar to CUDA programs. They interleave serial code consisting of C-like code and kernel calls consisting of SIMD code (single instruction multiple data) organized into thread blocks. However, all thread blocks of a given kernel start simultaneously and run concurrently. In addition, data transfer between global memory and shared memories occur only at the beginning or at the end of the execution of a thread block.

This latter two restrictions make algorithms analysis simpler. Given, an algorithm and an input data of size n, running our *ideal manycore machine*, we are interested in the *work* (that is, the total number of arithmetic operations performed by the streaming multiprocessors) the *span* (that is, the running time of the program ignoring the serial code and host-device data transfer) and the number of kernel calls. For these three measures, the lower the better.

3. Plain Multiplication on the GPU

Consider two univariate polynomials over a finite field

$$a = a_n x^n + \cdots + a_1 x + a_0 \text{ and } b = b_m x^m + \cdots + b_1 x + b_0, \text{ with } n \geq m. \tag{1}$$

We start from the so-called *long multiplication*[2], which computes the product $a \times b$ in the way we all learned integer multiplication in primary school. To parallelize it we proceed as follows.

- Multiplication phase: The set of terms $\{a_i b_j x^{i+j} \mid 0 \leq i \leq n, 0 \leq j \leq m\}$ forms a parallelogram that we decompose into small rectangles and triangles such that each of them can be computed by a thread block. Within a thread block, a thread computes all the terms of a given degree range and adds them up into a vector, where this *vector* is associated to the thread block.
- Addition phase: All the thread block vectors are added together by means of a parallel reduction.

Note that in this process, every coefficient of one polynomial is multiplied with every coefficient of the other polynomial. These products of coefficients are independent of each other and, thus, can be done in parallel essentially in $O(1)$ time (if we have sufficiently many processors and enough space to store mn coefficients). These coefficient products can be added in $log(m)$ parallel steps. So the overall complexity of this parallel algorithm is dominated by that of the addition phase. Though the algorithm that we just described seems not realistic due to the limited computing resources of today's computers, still we can adapt its principle in such a way that we make the best use of streaming multiprocessors of GPUs.

3.1. Implementation

In order to assign each thread block with the products $a_i b_j x^{i+j}$ that this thread block performs, we view $a_i b_j x^{i+j}$ as the point of coordinates (i, j) in an orthogonal Cartesian coordinate system, where the quadrant of positive coordinates is the "south-west region". Therefore, on Figure 1 the origin of this coordinate system is the right-top-most corner of the *parallelogram*, where this parallelogram is defined as the points of coordinates (i, j) satisfying $0 \leq i \leq n$ and $0 \leq j \leq m$.

[2] http://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication

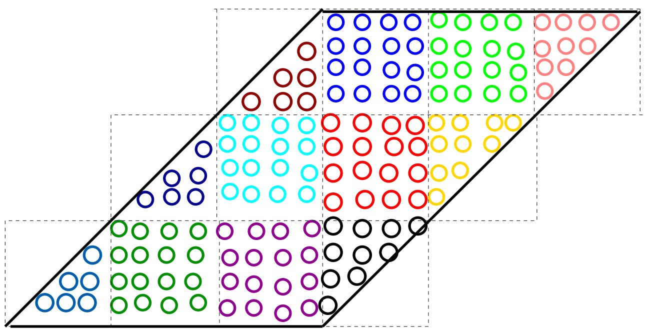

Figure 1. Dividing the work of coefficient multiplication among threadblocks.

Let r, t be two positive integers greater than 1. For simplicity let us assume that $m \geq r$ and $n \geq t$ hold. Moreover, we assume that r divides $m+1$ and t divides $n+1$.

The mapping between the parallelogram and the thread blocks is precisely defined as follows.

(i) We partition the parallelogram horizontally into $(m+1)/r$ parallelograms of height r (and basis $n+1$) that we call *long parallelograms*. Moreover, we attach to the ℓ-th long parallelogram a vector V_ℓ of length $n+r$, that we call the ℓ-th *long vector*.

(ii) We partition each long parallelogram vertically into $(n+1)/t$ equally spaced *small trapezoids*. More precisely, within a long parallelogram, the left-most and right-most trapezoids are triangles while all other trapezoids are rectangles. Observe that for each small trapezoid, all points with the same i-coordinate, say k, correspond to terms of degree k.

On Figure 1, we have $n = m = 11$. The original parallelogram is horizontally decomposed into $r = 3$ long parallelograms and each of those is decomposed into $t = 4$ rectangles/triangles.

Each thread block is in charge of computing all products within a given small trapezoid (rectangle or triangle). Let us assume for simplicity that t is a multiple of the number T of threads within a thread block. Define $s := t/T$.

Within a thread block of the ℓ-th long parallelogram, a thread is in charge of computing all terms of degrees $k, k+1, \ldots, k+s-1$, for some k, and adding the terms of degree $d \in \{k, k+1, \ldots, k+s-1\}$ in $V_\ell[d]$.

We now proceed with the addition phase. The vectors $V_0, V_1, \ldots, V_w$, with $w = (m+1)/r$, are added such that terms of the same degree are added together. This is done through a parallel reduction in $\Theta(\log(w))$ parallel steps.

3.2. Parallelism

We analyze the work and span of this algorithm. We start with the multiplication phase. We have $\frac{m+1}{r}\frac{n+1}{t}$ thread blocks. Each thread block has a work of $t(2r-1)$ and a span of $s(2r-1)$. Thus the work is $\Theta(n^2)$ while the parallelism is $\Theta(n^2/sr)$. For our GPU card, we choose $r = t = 2 * 9$ and $s \in \{4, 6\}$. Next, we proceed with the addition phase. The work is essentially is $2nw$, with $w = (m+1)/r$ and the span is in $\Theta(\log(w))$. Thus, the overall parallelism is $O(n^2/\log(m))$.

3.3. Experimental Results

We have experimented the CUDA implementation of the plain univariate multiplication described in the previous section. We use both *balanced* and *unbalanced* pairs of polynomials, see Table 1 and 2 respectively. By balanced, following [21], we mean a pair of univariate polynomials of equal degree, which is a favorable case for optimized FFT-based polynomial multiplication.

degree	GPU Plain multiplication	GPU FFT-based multiplication
2^{10}	0.00049	0.0044136
2^{11}	0.0009	0.004642912
2^{12}	0.0032	0.00543696
2^{13}	0.01	0.00543696
2^{14}	0.045	0.00709072

Table 1. Comparison between plain and FFT-based polynomial multiplications for balanced pairs ($n = m$) on CUDA.

n	m	GPU Plain multiplication
2^{10}	2^8	0.00041
2^{11}	2^8	0.0005
2^{11}	2^{10}	0.00073
2^{12}	2^8	0.00057
2^{12}	2^{10}	0.0011
2^{13}	2^8	0.00074
2^{13}	2^{10}	0.0018
2^{13}	2^{12}	0.0061
2^{14}	2^8	0.0010
2^{14}	2^{10}	0.0031
2^{14}	2^{12}	0.011
2^{14}	2^{13}	0.02

Table 2. Computation time for plain multiplication on CUDA for unbalance pairs ($n \neq m$).

In Table 1, we compare the computation time of our CUDA based implementation of parallel plain multiplication with the highly optimized FFT-based multiplication in CUDA reported in [19]. Our implementation outperforms that of FFT-based multiplication until the degree 2^{12}.

FFT-based multiplication may not perform well for unbalanced pairs, see [21] for details. In plain multiplication, this is not true. Computation times for plain multiplication on CUDA of unbalanced pairs are reported in Table 2.

4. Plain Division on the GPU

Consider again two univariate polynomials over a finite field

$$a = a_n x^n + \cdots + a_1 x + a_0 \quad \text{and} \quad b = b_m x^m + \cdots + b_1 x + b_0, \quad \text{with} \quad n \geq m. \tag{2}$$

The only opportunity for concurrent execution is within each division step. With the above notations, the first division step computes

$$a' \ \leftarrow a - \frac{a_n}{b_m} x^{n-m} b, \tag{3}$$

which can be viewed as a *Gaussian elimination step*. Assuming that this is done by several thread blocks, the next division step requires to broadcast the leading coefficient of the intermediate remainder a' to all thread blocks, which is a severe performance bottleneck.

Our solution consists of letting each thread block compute the leading coefficients of s consecutive intermediate remainders, for a well chosen integer s. In this way, each thread block computes a coefficient segment (of size $2\,s$) of s consecutive intermediate remainders without synchronization. Though this increases the total work by (at most) a $\frac{1}{3}$ factor, this improves performances significantly.

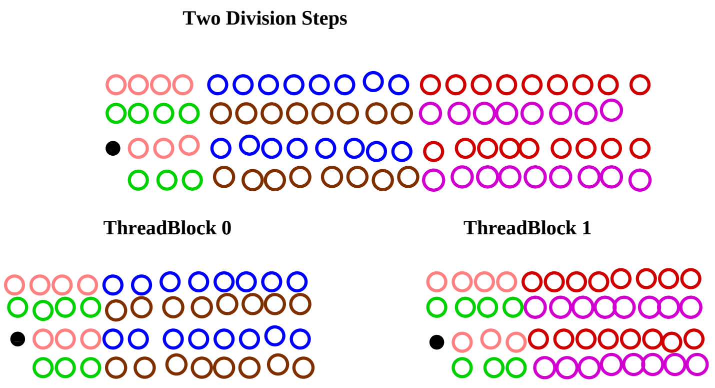

Figure 2. Each threadblock will work with a^+ (resp. b^+) along with a segment of coefficients of length $2s$.

4.1. Implementation

Recall that s is a fixed positive integer. Let a^+ (resp. b^+) be the sequence of the s *most significant* terms (that is, non-zero terms of largest degrees) of both polynomials a and b. Each thread block loads a^+ and b^+ in shared memory such that it can compute s consecutive coefficients of the quotient in the division of a by b, without synchronizing with other thread blocks.

The other coefficients of the polynomial a are divided into $(n - s)/(2s)$ segments (of consecutive, zero or non-zero, coefficients). For simplicity, let assume that $2s$ divides $n - s$.

Each thread block reads a segment of $2s$ coefficients of a and is responsible for updating these coefficients after each division step during the execution of the thread block. As each thread block can perform s division steps independently, it will also require a segment of $3s$ coefficients from the polynomial b. This is, indeed, $3s$ (and not $2s$) which is required from b since after division step the degree of the current remainder, namely a, reduces at least by 1. Overall, each thread block loads $7s$ coefficients in the shared memory.

With today's GPU cards, the maximum number of threads in a thread block is roughly one of order of magnitude away from the number of machine words that can be stored in shared memory. So we keep s in the order of the maximum number of threads in a thread block supported by a typical GPU card. As a consequence, each thread is responsible for updating $O(1)$ number of coefficients of a.

On Figure 2, $s = 4$. ThreadBlock0 is responsible for updating the a^+ part along with other 8 consecutive coefficients. In the same way, ThreadBlock1 is responsible for updating the a^+ part along with other 8 consecutive coefficients.

4.2. Parallelism

In the worst case, we need $n - m + 1$ division steps. Since one kernel call will reduce the degree of the current remainder at least by s, we have $(n - m + 1)/s$ kernel calls at most. During one kernel call, one thread is doing $O(s)$ operations. So the span of the whole procedure can be bounded by $O(n-m)$ or $O(n)$. The work is $O(m(n-m))$ or $O(nm)$. So the parallelism is $O(m)$.

4.3. Experimental Results

We have compared the running time of our CUDA implementation of the plain division with the serial C implementation of the fast division (see Chapter 9 in [10]) from the NTL [25] library. The latter algorithm is based on FFT techniques and its work fits within $O(d\log(d)\log(\log(d)))$ coefficient operations, with $d = \max(m, n)$. The input polynomials used in our experimentation are dense random (univariate) polynomials with coefficients in a finite field whose characteristic

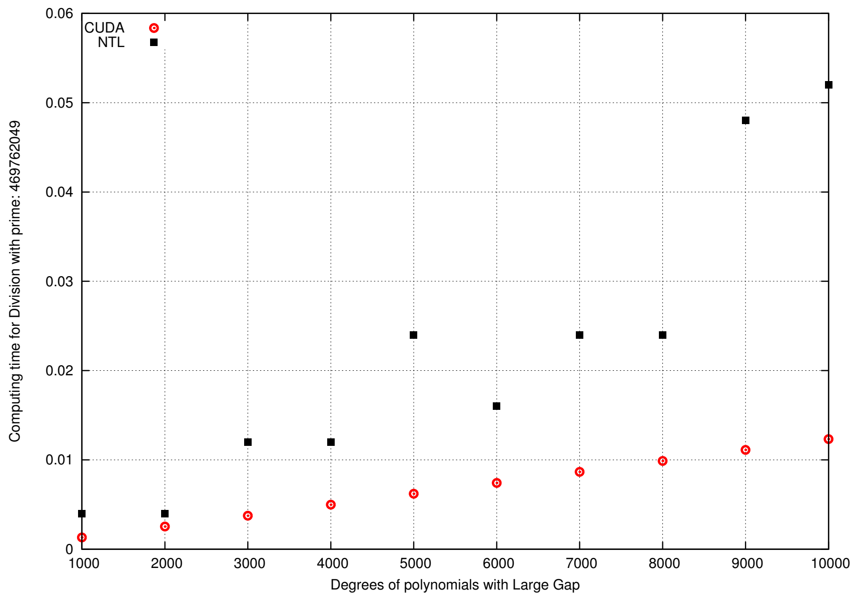

Figure 3. Comparison between parallel plain division on CUDA and fast division in NTL for univariate polynomials with large degree gap.

is a machine word prime. We use the following primes: 7, 9001 and 469762049. Our GPU code does not depend on the prime while NTL uses different algorithms depending on the prime. For a given degree pattern, the NTL running time varies at most by a factor of 2 from one of our primes to another. The degrees of our input polynomials satisfy $n = 2m$. The running time of our CUDA code outperforms that of NTL by a factor from 3 to 5, for $1,000 \leq n \leq 10,000$, see Figure 3.

5. The Euclidean Algorithm on the GPU

Recall that a and b designate two univariate polynomials over a finite field, such that either b is zero or the degree of b is not greater than that of a. The Euclidean Algorithm computes the GCD of (a, b) in the following way.

(i) if $b = 0$, then return a,

(ii) if $b \neq 0$, then return the GCD of (b, r) where r is the remainder in the division of a by b.

As for the plain division, we have no other choices than parallelizing each division step. Moreover, in order to minimize data transfer we will again let each thread block work on several consecutive division steps without synchronizing. For this to be possible, each thread block within a kernel computes the *most significant* terms of the current pair *dividend-divisor*.

5.1. Implementation

Let $s > 1$ be an integer. Each kernel call replaces the polynomial pair (a, b) by a *GCD preserving pair* (a', b') (that is, a a pair of polynomials with the same GCD as (a, b)) such that we have

$$\max(\deg(a), \deg(b)) - \max(\deg(a'), \deg(b')) \ \geq \ s.$$

Moreover, each kernel call performs at most s division steps.

- If initially $\mid \deg(a) - \deg(b) \mid \geq \ s$ holds, we simply use our kernel for s division steps with a fixed divisor.

- If initially $\mid \deg(a) - \deg(b) \mid < \ s$ holds, we modify our division kernel as follows: each thread block reads the s most significant terms for both a and b and; in addition, it reads a segment of $3s$ coefficients for both a and b. Indeed, during such a kernel call, the roles of a and b, as dividend or divisor, can be exchanged.

5.2. Parallelism

We first consider the work of the algorithm sketched above. Since $n \geq m$, the number of kernel calls is at most $\lceil \frac{n}{s} \rceil$. The number of thread blocks per kernel call is at most $\lceil \frac{n}{2s} \rceil$. The number of arithmetic operations per thread block is at most $6s^2$. Thus the work is in $O(n^2)$, as expected. However, there is an increase of work w.r.t. a serial GCD computation by a constant factor in order to reduce the amount of synchronization. Moreover, there is an increase of memory consumption w.r.t. the GPU division computation by a constant factor due to the case where the degree gap between a and b is less than s. Now, we consider the span. Since the number of kernel calls is at most $\lceil \frac{n}{s} \rceil$, and since the number of division steps per kernel call is at most s, the span is in $O(n)$, which as good as in the case of systolic arrays [2].

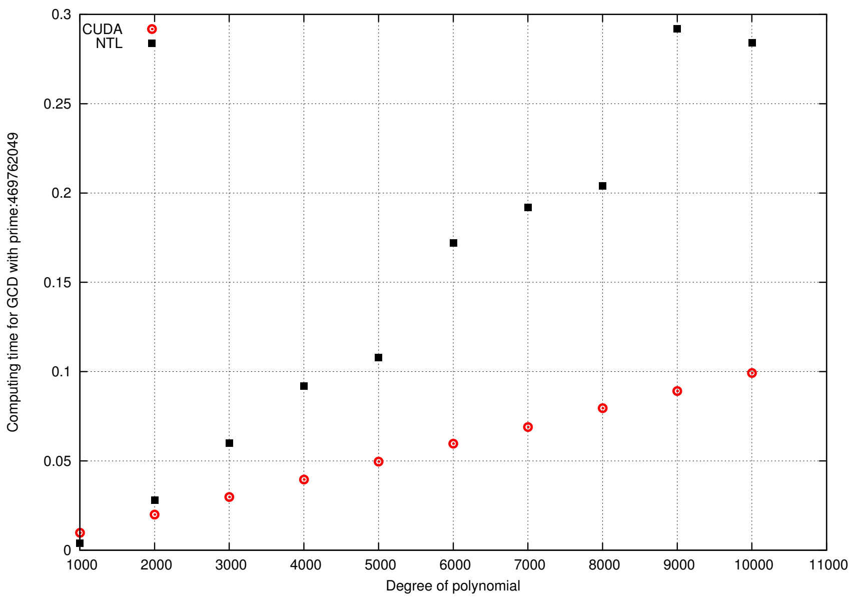

Figure 4. Comparison between parallel GCD on CUDA and FFT-based GCD in NTL for univariate polynomials, with the same degree ($n = m$).

5.3. Experimental Results

We have compared the running time of our CUDA implementation of the Euclidean Algorithm with the serial C implementation of the Half-GCD algorithm (see Chapter 11 in [10]) from the NTL [25] library. The latter algorithm is based on FFT techniques and its work fits within $O(d\log(d)\log(\log(d)))$ coefficient operations, whale that of the former algorithm amounts to $O(d^2)$ coefficient operations, for input polynomials of degree d.

As for the experimentation with the plain division, the input polynomials used in our experimentation are dense random (univariate) polynomials with coefficients in a finite field whose characteristic is a machine word prime. Here again, we use the primes 7, 9001, 469762049 and we observe that our GPU code does not depend on the prime while NTL uses different algorithms depending on the prime. For a given degree pattern, the NTL running time varies at most by a factor of 2 from one of our primes to another.

n	m	GCD on CUDA with s = 512	GCD on CUDA with s = 0
1000	500	0.010	0.024
2000	1500	0.024	0.058
3000	2500	0.039	0.108
4000	3500	0.053	0.158
5000	4500	0.069	0.203
6000	5000	0.056	0.235
7000	6000	0.066	0.282
8000	7000	0.076	0.324
9000	8000	0.087	0.367
10000	9000	0.097	0.411

Table 3. GCD implementation on CUDA with two different values of s.

Figure 4 correspond to 469762049. Indeed, we are interested in large primes since they support modular methods for polynomial system solving [6].

As reported by Figure 4, our implementation is almost three times faster than that of NTL for polynomials whose degrees range between 1,000 and 10,000. Recall that this degree range is also what is of interest for the same purpose of polynomial system solving.

The technique of computing s leading coefficients in every thread block, implemented in both division and GCD algorithm, has two major advantages. First, it reduces the number of kernel calls by a factor of s. Second, it reduces the amount of memory transfer between the global and shared memory by a factor of s.

In Table 3, we compare the computation time between two versions of our CUDA implementation of the Euclidean Algorithm. The first one sets $s = 512$ and the other one does not use at all this technique of computing s leading coefficients in every thread block. In this latter implementation, leading coefficients are kept up-to-date in the global memory such that they can be accessed by every thread block. Thus, in this scheme, every thread block works on a single division step between two updates of the leading coefficient in the global memory.

As mentioned above, the former implementation increases the work but reduces parallelization overheads in a significant manner. Table 3, shows that the former method outperforms the latter by a speedup factor varying from 2 to 4.

6. Conclusion

Motivated by the implementation of polynomial system solvers over finite fields, we were lead to parallelize plain univariate polynomial arithmetic on GPUs. For the degree range that we are targeting, namely from 2^{10} up to 2^{12}, our GPU code for plain multiplication outperforms our GPU code for FFT-based multiplication. For the degree range $2^{10} \cdots 2^{18}$, our GPU code for computing polynomial GCDs via the Euclidean Algorithm runs in linear time w.r.t the maximum degree of the input polynomials. Such sizes are sufficient for many applications.

We observed that controling parallelization overheads (synchronization on data via global memory, number of kernel calls, etc.) was essential for reaching peak performance in our implementation.

References

[1] R. A. Arce-Nazario, E. Orozco and D. Bollman. Reconfigurable hardware implementation of a multivariate polynomial interpolation algorithm. *Int. J. Reconfig. Comput.*, vol. 2010, pages 2:1–2:14, 2010.

[2] R. P. Brent, H. T. Kung, and F. T. Luk. Some Linear-time Algorithms for Systolic Arrays. In *IFIP Congress*, pages 865–876, 1983.

[3] M. F. I. Chowdhury, M. Moreno Maza, W. Pan, and É. Schost. Complexity and Performance Results for non FFT-based Univariate Polynomial Multiplication. In *Proc. of Advances in mathematical and computational methods: addressing modern of science, technology, and society*, AIP conference proceedings, volume 1368, pp. 259-262, 2011.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and Clifford Stein, *Introduction to Algorithms (3. ed.)* MIT Press, 2009.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest and Clifford Stein, Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company, 2001.

[6] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.

[7] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. In Proceedings of the IEEE Special issue on *Program Generation, Optimization, and Platform Adaptation*, volume 93, number 2, pages 216–231, 2005.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, USA, October 1999.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN*, 1998.

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

[11] J. von zur Gathen and J. Shokrollahi. Efficient FPGA-based karatsuba multipliers for polynomials over F_2. In *Proceedings of the 12th international conference on Selected Areas in Cryptography*, SAC'05, pages 359–369, Springer-Verlag, 2006.

[12] S. A. Haque and M. Moreno Maza. *Determinant Computation on the GPU using the Condensation Method* , volume 341. J. of Physics: Conference Series, 2012.

[13] M. A. Hasan and V. K. Bhargava. Bit-Serial Systolic Divider and Multiplier for Finite Fields GF(2^m). *IEEE Trans. Comput.*, vol. 41, num. 8, pages 972–980, IEEE Computer Society, 1992.

[14] J. Hong and H. T. Kung. I/O Complexity: The Red-blue Pebble Game. In *STOC*, pages 326–333, 1981.

[15] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition.* Addison-Wesley Professional, 1997.

[16] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley Professional, 1997.

[17] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching 2nd Edition.* Addison-Wesley Professional, 1998.

[18] H. T. Kung and C. L. Leiserson. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, ed. C. Mead and L. Conway, Addison-Wesley, Reading, MA, 1980, pp. 271-292.

[19] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a GPU. In *J. of Physics: Conference Series*, volume 256, 2010.

[20] M. Moreno Maza and W. Pan. Solving Bivariate Polynomial Systems on a GPU. In *J. of Physics: Conference Series*, volume 341, 2011.

[21] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. In *Int. J. Found. Comput. Sci.*, volume 22, numbe 5, pages 1035-1055, 2011.

[22] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In D. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCS*, Heidelberg, 2010. Springer-Verlag Berlin.

[23] M. Nele, S. B. Örs, B, Preneel and J. Vandewalle An FPGA Implementation of a Montgomery Multiplier Over GF(2^m). *J. of Computers and Artificial Intelligence*, vol. 23, num. 5, pages 487-499, 2004.

[24] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms, In Proceedings of the IEEE, special issue on *Program Generation, Optimization, and Adaptation*, volume 93, numbe 2, pages 232-275, 2005.

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

[26] C. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1993.