Dense Arithmetic over Finite Fields with the CUMODP Library

Sardar Anisul Haquen[1], Xin Li[2], Farnam Mansouri[1], Marc Moreno Maza[1], Wei Pan[3], and Ning Xie[1]

[1] University of Western Ontario, Canada
{shaque4,fmansou3,moreno,nxie6}@csd.uwo.ca
[2] Universidad Carlos III, Spain
xli@inf.uc3m.es
[3] Intel Corporation, Canada
wei.pan@intel.com

Abstract. CUMODP is a CUDA library for exact computations with dense polynomials over finite fields. A variety of operations like multiplication, division, computation of subresultants, multi-point evaluation, interpolation and many others are provided. These routines are primarily designed to offer GPU support to polynomial system solvers and a bivariate system solver is part of the library. Algorithms combine FFT-based and plain arithmetic, while the implementation strategy emphasizes reducing parallelism overheads and optimizing hardware usage.

Keywords: Polynomial arithmetic, parallel processing, many-core GPUs

1 Overview

Polynomial multiplication and matrix multiplication are at the core of many algorithms in symbolic computation. Expressing, in terms of multiplication time, the algebraic complexity of an operation like univariate polynomial division or the computation of a characteristic polynomial is a standard practice, see for instance the landmark book [4]. At the software level, the motto "reducing everything to multiplication"[4] is also common, see for instance the computer algebra systems Magma[5] [1], NTL[6] or FLINT[7].

[4] Quoting a talk title by Allan Steel, from the Magma Project.
[5] Magma: http://magma.maths.usyd.edu.au/magma/
[6] NTL: http://www.shoup.net/ntl/
[7] FLINT: http://www.flintlib.org/

With the advent of hardware accelerator technologies, multi-core processors and Graphics Processing Units (GPUs), this reduction to multiplication is, of course, still desirable, but becomes more complex since both algebraic complexity and parallelism need to be considered when selecting and implementing a multiplication algorithm. In fact, other performance factors, such as cache usage or CPU pipeline optimization, should be taken into account on modern computers, even on single-core processors. These observations guide the developers of projects like SPIRAL[8] [16] or FFTW[9] [3].

The CUMODP library provides arithmetic operations for dense matrices and dense polynomials primarily with modular integer coefficients, targeting many-core GPUs. Some operations are available for integer or floating point coefficients as well. A large portion of the CUMODP library code is devoted to polynomial multiplication and the integration of that operation into higher-level algorithms.

Typical CUMODP operations are matrix determinant computation, polynomial multiplication (both plain and FFT-based), univariate polynomial division, the Euclidean algorithm for univariate polynomial GCDs, subproduct tree techniques for multi-point evaluation and interpolation, subresultant chain computation for multivariate polynomials, bivariate system solving. The CUMODP library is written in CUDA [15] and its source code is publicly available at `www.cumodp.org`.

In this note, we give an overview of the implementation techniques of the CUMODP library. In Section 2, we discuss a model of multithreaded computation, combining fork-join and single-instruction-multiple-data parallelisms, with an emphasis on estimating parallelism overheads of programs written for modern many-core architectures. For each key routine of the CUMODP library this model is used to minimize parallelism overheads by determining an appropriate value range for a given program parameter, e.g. number of threads per block. Experimentation confirms the effectiveness of this model.

Secondly, the design of the CUMODP library emphasizes the importance of adaptive algorithms in the context of many-core GPUs, see Section 3. that is, algorithms which adapt their behavior according to the available computing resources. Based on these techniques, we have obtained the first GPU implementation of subproduct tree techniques for multi-point evaluation and interpolation of univariate polynomials. Hence we compare our code against probably the best serial C code, namely the FLINT library, for the same operations. For sufficiently large input data and on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30.

We conclude in Section 4 by presenting an application of the CUMODP library to bivariate system solving.

[8] `http://www.spiral.net/`
[9] `http://www.fftw.org/`

2 A many-core machine model for designing algorithms with minimum parallelism overheads

Our model of multithreaded computation [9] extends the following previous works for which we summarize key features and limitations. The PRAM (parallel random access machine) model [5] supports data parallelism but not task parallelism. Moreover, this model cannot support memory traffic issues like cache complexity and memory contention. The Queue Read Queue Write PRAM [6] considers memory contention, however, it unifies in a single quantity time spent in arithmetic operations and time spent in read/write accesses. We believe that this unification is not appropriate for recent many-core processors, such as GPUs, for which the ratio between one global memory read/write access and one floating point operation can be in the 100's. The TMM (Threaded Many-core Memory) model [12] retains many important characteristics of GPU-type architectures, however, the running time estimate on P cores is not given by a Graham-Brent theorem [7]. We believe that, for the purpose of code optimization, this latter theorem is an essential tool.

Our proposed *many-core machine model* (MMM) aims at optimizing algorithms targeting implementation on GPUs. Our abstract machine possesses an unbounded number of streaming multiprocessors (SMs). However, each SM has a finite number of processing cores and a fixed-size local memory. An MMM machine has a two-level memory hierarchy, comprising an unbounded global memory with high latency and low throughput while SMs local memories have low latency and high throughput. Similarly to a CUDA program, an MMM program specifies for each kernel the number of thread-blocks and the number of threads per thread-block. An MMM machine has two parameters:

U: time (expressed in clock cycles) to transfer one machine word between the global memory and the local memory of any SM,
Z: size (expressed in machine words) of the local memory of any SM.

An MMM program $\mathcal{P}$ is a directed acyclic graph (DAG), called the *kernel DAG*, whose vertices are kernels and edges indicate serial dependencies. Since each kernel of the program $\mathcal{P}$ decomposes into a finite number of thread-blocks, we map $\mathcal{P}$ to a second graph, called the *thread-block DAG* of $\mathcal{P}$, whose vertex set consists of all thread-blocks of $\mathcal{P}$. We consider three complexity measures:

- the *work* $W(\mathcal{P})$, which is the total number of local operations (arithmetic operation, read/write requests in the local memory) performed by all threads,
- the *span* $S(\mathcal{P})$, which is the longest path, counting the weight (span) of each vertex (kernel), in the kernel DAG,
- the *parallelism overhead* $O(\mathcal{P})$, which is the total data transfer time (between global and local memories) of all its kernels.

Using these complexity measures, we derive a Graham-Brent theorem with parallelism overhead.

Theorem 1. *Let K be the maximum number of thread blocks along an anti-chain of the thread-block DAG of $\mathcal{P}$. Then the running time $T_{\mathcal{P}}$ of the program $\mathcal{P}$ satisfies:*

$$T_{\mathcal{P}} \leq (N(\mathcal{P})/\mathsf{K} + L(\mathcal{P}))\, C(\mathcal{P}). \tag{1}$$

where $N(\mathcal{P})$, $L(\mathcal{P})$ and $C(\mathcal{P})$ are respectively: the number of vertices in the thread-block DAG, the critical path length (where length of a path is the number of edges in that path) in the thread-block DAG and the maximum running time of local operations by a thread among all the thread-blocks.

We have applied the MMM model for optimizing the CUDA implementation of operations like plain univariate polynomial division, plain univariate polynomial multiplication and the Euclidean algorithm. for dense polynomials over small prime fields. In each case, a program $\mathcal{P}(s)$ depends on a parameter s which varies in a range $\mathcal{S}$ around an initial value s_0, such that the work ratio W_{s_0}/W_s remains essentially constant meanwhile the parallelism overhead O_s varies more substantially, say $O_{s_0}/O_s \in \Theta(s - s_0)$. Then, we determine a value $s_{\min} \in \mathcal{S}$ maximizing the ratio O_{s_0}/O_s. Next, we use our version of Graham-Brent theorem to check whether the upper bound for the running time of $\mathcal{P}(s_{\min})$ is less than that of $\mathcal{P}(s_o)$. If this holds, we view $\mathcal{P}(s_{\min})$ as a solution of our problem of algorithm optimization (in terms of parallelism overheads).

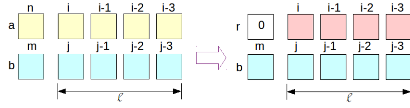

Fig. 1. Naive division algorithm of a thread-block with $s = 1$: each kernel performs 1 division step

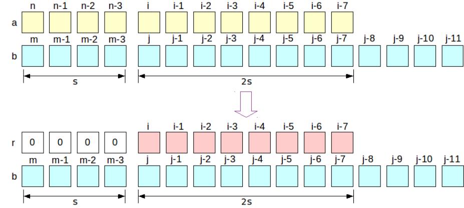

Fig. 2. Optimized division algorithm a thread-block with $s > 1$: each kernel performs s division steps

For each operation, the program parameter s controls the amount of work and parallelism overheads of a thread-block. Figures 1 and 2 illustrate the role of this parameter in our implementation of plain division. See [9] for details.

Applying the optimization strategy described above lead us to determine an optimum value of s among those implied by constraints like the size of the local memory Z or the data transfer time U. For plain polynomial multiplication, this analysis suggested to minimize s which was verified experimentally, as illustrated by Figure 3. For the Euclidean algorithm, our analysis suggested to maximize the program parameter s, which was again verified experimentally, as illustrated by Figure 4. Our experimental results were obtained on a GPU card NVIDIA Tesla C2050.

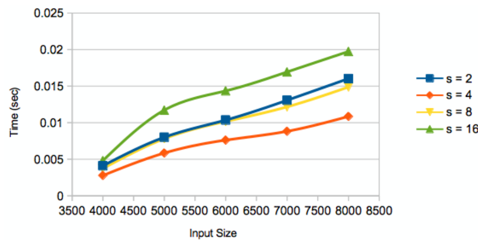

Fig. 3. Plain polynomial multiplication: varying the program parameter s

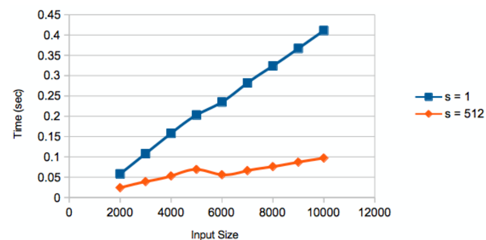

Fig. 4. Euclidean algorithm: varying the program parameter s

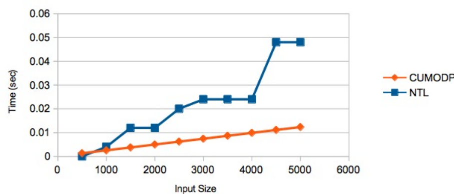

Fig. 5. CUMODP plain polynomial division vs NTL FFT-based (asymptotically fast) polynomial division.

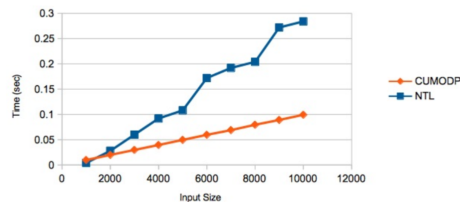

Fig. 6. CUMODP plain Euclidean algorithm vs NTL FFT-based polynomial GCD.

Figures 5 and 6 show that the optimized CUMODP implementation of the plain division and the Euclidean algorithm outperforms the NTL implementation of the FFT-based plain division and polynomial GCD computation. Of course, CUMODP code is multithreaded while NTL code is serial. On the other hand, NTL uses asymptotically fast algorithms. The key observation is that optimized implementation of multithreaded plain algorithms provide useful alternative to any serial code. In fact, as we will see in the next section, multithreaded plain algorithms play an essential in higher-level applications targeting many-core GPUs.

3 Adaptive algorithms

Up to our knowledge, the CUMODP library offers the first GPU implementation of subproduct tree techniques [4][Chapter 10] for multi-point evaluation and interpolation of univariate polynomials. The parallelization of those techniques raises the following challenges on hardware accelerators.

1. The divide-and-conquer formulation of operations on subproduct-trees is not sufficient to provide enough parallelism and one must also parallelize the underlying polynomial arithmetic operations, in particular polynomial multiplication.
2. During the course of the execution of a subproduct tree operation (construction, evaluation, interpolation), the degrees of the involved polynomials

vary greatly; thus, so does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.

To address the first challenge on many-core GPUs, we combine *parallel plain arithmetic* and *parallel fast arithmetic*. For the former we rely on [8] and, for the latter we extend the work of [13]. Indeed, parallel fast arithmetic alone would not suffice to provide good speedup factors since subproduct tree operations require lots of calculations with low-degree polynomials.

To address the second challenge, we employ *adaptive algorithms*. That is, algorithms that adapt their behavior according to the available computing resources. For instance, each plain multiplication is performed by a single streaming multiprocessor (SM), since plain arithmetic is used for input polynomials of small sizes. Meanwhile, each FFT-based multiplication is computed by a kernel call, thus using several SMs. In fact, this kernel computes a number of FFT-based products concurrently.

	Evaluation			Interpolation		
Deg.	CUMODP	FLINT	SpeedUp	CUMODP	FLINT	SpeedUp
2^{12}	0.1361	0.02	0.1468	0.1671	0.03	0.1794
2^{13}	0.1580	0.07	0.4429	0.1963	0.09	0.4584
2^{14}	0.2034	0.17	0.8354	0.2548	0.22	0.8631
2^{15}	0.2415	0.41	1.6971	0.3073	0.53	1.7242
2^{16}	0.3126	0.99	3.1666	0.4026	1.26	3.1294
2^{17}	0.4285	2.33	5.4375	0.5677	2.94	5.1780
2^{18}	0.7106	5.43	7.6404	0.9034	6.81	7.5379
2^{19}	1.0936	12.63	11.5484	1.3931	15.85	11.3768
2^{20}	1.9412	29.2	15.0420	2.4363	36.61	15.0268
2^{21}	3.6927	67.18	18.1923	4.5965	83.98	18.2702
2^{22}	7.4855	153.07	20.4486	9.2940	191.32	20.5851
2^{23}	15.796	346.44	21.9321	19.6923	432.13	21.9441

Table 1. Multi-point evaluation and interpolation: FLINT vs CUMODP.

To evaluate our implementation of subproduct tree techniques, we measured the effective memory bandwidth of our GPU code for parallel multi-point evaluation and interpolation on a card with a theoretical maximum memory bandwidth of 148 GB/S, our code reaches peaks at 64 GB/S. Since the arithmetic intensity of our algorithms is high, we believe that this is a promising result.

All implementation of subproduct tree techniques that we are aware of are pure serial code. This includes [2] for $GF(2)[x]$, the FLINT library[10] and the `Modpn` library [11]. Hence we compare our code against probably the best serial C code (namely the FLINT library). For sufficiently large input data, running on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30. Experimental data can be found in Table 1.

4 Application

In [14], two of the co-authors of this note reported on the implementation of a bivariate polynomial system solver (based on the theory of *regular chains* and working with coefficients in small prime fields) partially written in CUDA and partially written in C. In that implementation, polynomial subresultant chains were calculated in CUDA while univariate polynomial GCDs were computed in C either by means of the plain Euclidean algorithm or an asymptotically fast algorithm).

The authors observed that about 90% of the overall running time of their solver was spent in univariate GCD computations. They also noted that most of these GCD calculations were using the plain algorithm since the degrees of the input polynomials were not large enough for using the FFT-based algorithm.

System	Pure C	Mostly CUDA code	SpeedUp
dense-70	5.22	0.50	10.26
dense-80	6.63	0.77	8.59
dense-90	8.39	1.16	7.19
dense-100	19.53	1.80	10.79
dense-110	21.41	2.57	8.33
dense-120	25.71	3.48	7.39
sparse-70	0.89	0.31	2.81
sparse-80	3.64	1.18	3.09
sparse-90	3.13	0.92	3.40
sparse-100	8.86	1.20	7.38

Table 2. Bivariate system solving over a small prime field: timings in sec.

These observations have lead to a CUDA implementation of the plain Euclidean algorithm which is reported in [8]. More recently, the same authors have put together in a single CUDA application the work reported in [14] and [8], leading to a bivariate polynomial system solver which is mostly written in CUDA. Table 2 compares this latter with an implementation of our bivariate system solver (presented in [14]) entirely written in C. Some of the input systems are random dense and the others are sparse. The number attached to each system name is the total degree of each input polynomial. For each system, the total number of solutions is essentially the square of that degree.

One can see that for a complex application like a polynomial system solver, a CUDA implementation can provide substantial benefit w.r.t. a pure C implementation. We should also point out that our CUDA implementation can be further improved. In particular, the top-level algorithm is still implemented in C and lots of data transfers are still taking place between the host (CPU) and the device (GPU). This performance bottleneck can be removed by using the latest programming model of CUDA.

References

1. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).
2. R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in gf(2)[x]. In *Proceedings of the 8th international conference on Algorithmic number theory*, ANTS-VIII'08, pages 153–166, Berlin, Heidelberg, 2008. Springer-Verlag.
3. M. Frigo and S. G. Johnson. The design and implementation of FFTW3. 93(2):216–231, 2005.
4. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.
5. P. B. Gibbons. A more practical PRAM model. In *Proc. of SPAA*, pages 158–168, 1989.
6. P. B. Gibbons, Y. Matias, and V. Ramachandran. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. on Comput.*, 28(2):733–769, 1998.
7. R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. on Applied Mathematics*, 17(2):416–429, 1969.
8. S. A. Haque and M. Moreno Maza. Plain polynomial arithmetic on GPU. In *J. of Physics: Conf. Series*, volume 385, page 12014. IOP Publishing, 2012.
9. S. A. Haque, M. Moreno Maza and N. Xie. A Many-core Machine Model for Designing Algorithms with Minimum Parallelism Overheads. In *Computing Research Repository*, vol. abs/1402.0264, 2014. `http://arxiv.org/abs/1402.0264`
10. W. B. Hart. Fast library for number theory: An introduction. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010, Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 88–91. Springer, 2010.
11. X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comput.*, 46(7):841–858, July 2011.
12. L. Ma, K. Agrawal, and R. D. Chamberlain. A memory access model for highly-threaded many-core architectures. In *Proc. of ICPADS*, pages 339–347, 2012.
13. M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a gpu. *J. of Physics: Conference Series*, 256, 2010.
14. M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a gpu. *J. of Physics: Conference Series*, 341, 2011.
15. J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
16. 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. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.