Hardware acceleration technologies (multicore processors, graphics processing units (GPUs), field-programmable gate arrays) provide vast opportunities for innovation in scientific computing. For tasks involving well-structured communication patterns, general-purpose computation on graphics hardware now allows several types of calculations (simulation, stock exchange data analysis, etc.) to satisfy the constraints of real time computing.

This proposal aims at developing techniques that will permit mathematical software to take great advantage of hardware acceleration technologies. Our focus is on *heterogeneous GPU-multicore architectures* and their support for *polynomial system solvers* based on *symbolic computation* for which specific features (intermediate expression swell, irregular data patterns, all of which leading to less structured and predictable communication requirements) make this proposal even more challenging.

1 Recent Progress

SOLVING SYSTEMS OF NON-LINEAR, ALGEBRAIC OR DIFFERENTIAL POLYNOMIAL EQUATIONS, is a fundamental problem in mathematical sciences. It has been studied for centuries and still stimulates many research developments. Today, it is a driving subject for the area of *symbolic computation*, also known as *computer algebra*. In each of the major computer algebra software packages (AXIOM, MAGMA, MAPLE, MATHEMATICA, REDUCE, etc.) various commands aim at solving all types of systems of equations relevant to symbolic computation. These symbolic solvers are powerful tools in scientific computing: they are well suited for problems where the desired output must be *exact* and they have been applied successfully in mathematics, physics, engineering, chemistry and education, with important outcomes. See Chapter 3 in [21] for an overview of these applications. While the existing computer algebra software packages have met with some practical success, symbolic computation is still under-utilized in areas like mathematical modeling and computer simulation. Part of this is due to the fact that much more complex computations are required - often beyond the scope of existing software. The implementation of symbolic solvers is, indeed, an extremely difficult task and requires techniques going far beyond the algebraic manipulation of equations; these include efficient memory management and parallel processing. The present proposal deals precisely with these latter aspects,

IN LESS THAN A DECADE, multicore and GPU architectures have brought respectively parallelism and supercomputing to the masses. Several areas of scientific computing (numerical linear algebra [15, 30], digital signal processing [20, 36, 46], etc.) have capitalized on this technological revolution. The commercialized software MATLAB (specialized in numerical linear algebra) with its Parallel Computing Toolbox [1] provides programming support, including library functions, that takes advantage of GPUs. This contrasts sharply with the state of affairs in symbolic computation. In fact, the use of GPUs in a major computer algebra like MAPLE is very limited. Nevertheless, a few research articles have been published on the subject: most of them are in the proceedings of PASCO'07 [40] and PASCO'10 [39], both of which I co-organized.

THE RESULTS OF MY CURRENT DISCOVERY GRANT, as planned, have substantially contributed to the interaction between high-performance computing and computer algebra. First, we have designed fast algorithms (mainly low-level routines) in support of polynomial system solvers [32] together with publicly available software packages implementing those algorithms [33]. Secondly, we have obtained, for those algorithms, efficient parallel counterparts, with multicore implementation [44, 43]. We have also investigated other implementation techniques, such as code generation [35] and code optimization for data locality [24, 31]. Thirdly, we have improved high-level algorithms of polynomial system solvers [11, 9]. As a result, the `solve` command of MAPLE relies today on those algorithms and their implementation realized in my research group. We have reported applications of those solvers to the

study of dynamical systems [10] and program verification [41].

WE HAVE ALSO OBTAINED UNANTICIPATED RESULTS, which are major tools for reaching high-performance. My research group has investigated implementation techniques on GPUs for various fundamental algorithms in symbolic computation: dense polynomial multiplication [37], the Euclidean Algorithm [23], dense linear algebra [22] and bivariate polynomial system solving [38]. This latter paper is a first step towards one of our objectives: developing polynomial system solvers capable of harvesting the horsepower of hardware acceleration technologies, in particular GPUs.

AT THE END OF THIS FIVE-YEAR CYCLE, it became clear that multicores alone were not able to support the parallelization of polynomial system solvers. Indeed, on multicore architectures, scheduling costs are often a performance bottleneck on problems of small or average size, while those problems often occur as subproblems when solving polynomial systems. In fact, arithmetic operations on polynomials have many opportunities for highly-threaded fine-grained parallelism which make them more suitable for GPUs. I believe that the *combination* of GPUs (typically for dense polynomial arithmetic) and multicores (typically for sparse polynomial arithmetic and high-level algorithms) is the right platform to advance the efficiency of polynomial system solvers. This claim is supported by the preliminary results obtained with our GPU-based bivariate solver [38] and the `cumodp` library [37, 23].

2 Objectives

Computer algebraists have utilized GPUs to perform low-level routines, namely arithmetic operations on polynomials and matrices, with the most sophisticated level being bivariate system solving [5, 38]. In fact, bivariate system solving can be cast into the category of *kernels for polynomials and matrices* since solving a bivariate system reduces to solving a linear system thanks to *subresultant theory* [50].

In geometrical terms, solving a bivariate system typically means describing the intersection of two planar curves while solving a system in three variables typically means describing the intersection of three space surfaces. Clearly this latter problem is much more complex than the former. In particular, scheduling computer resources (usage of processors and memory) for solving a bivariate system can essentially be done with the simple knowledge of the size of the input system (see [38]), thus statically. In contrast, a complete algorithmic solution for solving a trivariate system will necessarily allocate computer resources dynamically, due to the large variety of possible geometrical configurations.

Since implementing an application on GPU implies a decomposition of this application into a series of kernel calls, most of the scheduling is done statically. Therefore, solving polynomial systems in three or more variables on GPUs is an algorithmic challenge. Other features of GPU programming (efficiently utilizing the large number of cores, managing the number of registers and amount of on-chip memory, hiding global memory latency) make this task even more difficult. At the same time, this task is a very attractive challenge since there is great hope that GPUs can allow symbolic solvers to meet the constraints of real-time computing for many polynomial systems of practical interest.

These observations lead us to propose the first two projects below. The motivation of the third one is twofold. First, it deals with classical applications of symbolic computation that more efficient solvers will make successful. Secondly, it deals with program parallelization and program verification which are important aspects of the other two projects.

In the sequel of this proposal, when discussing the programming, execution or memory models of GPUs, we will use the terminology (kernel, thread block, streaming multi-processor, etc.) of the Compute Unified Device Architecture (CUDA) [45]. By manycore, we mean either a GPU device or the integration of a multicore processor with GPU devices (thus sharing memory). To emphasize the latter case, we freely use the adjectives *hybrid* and *heterogeneous*.

Project 1: A hybrid manycore computing model for symbolic computation. In the fork-join multithreaded programming model [6], work, span and cache complexity are well-understood complexity measures for concurrent algorithms. However, capturing analytically the parallelism overheads (e.g. scheduling costs) that a concurrency platform imposes on an executing program is a crucial question which has received little attention in the literature. The objective of our first project is to develop analytic tools for estimating parallelism overheads of algorithms targeting standard concurrency platforms on multicore and manycore architectures. These tools will help with the type of static scheduling required by the implementation of high-level algorithms on GPUs, such as those of Project 2.

Project 2: Solving polynomial systems on GPUs. The objective of our second project is to deliver algorithms and code for polynomial system solvers that would perform all intensive computations on the device (GPU) while the host (CPU) would only execute the top-level algorithms. Ultimately, those solvers are aimed to exploit heterogeneous and distributed architectures.

Project 3: Program analysis. Two important application areas of symbolic computations are computer program optimization and computer program verification. Techniques like quantifier elimination (for instance, via the Fourier-Motzkin Algorithm in the linear case or via cylindrical algebraic decomposition in the non-linear case) can support program transformation (like automatic parallelization) and generation of polynomial loop invariants. Within the computer algebra system MAPLE, we have initiated a framework, called `ProgramAnalysis`, targeting those operations. The objective of our third project is to extend the functionality and improve the performance of this framework. We will support non-linear parameters in the *polyhedron model* for program transformation (a clear need for supporting GPU code) and treat them with advanced quantifier elimination techniques based on triangular decompositions of semi-algebraic systems. On the program verification front, we will design efficient algorithms (based on sparse interpolation) for generating equation and inequality invariants. Last but not least, the `ProgramAnalysis` framework will be boosted by the GPU support obtained by Project 2, while Project 2 will provide test cases for `ProgramAnalysis`. In the sequel of this proposal, we focus on the automatic parallelization side of this project. However, a work plan for its component on program verification, based on our papers [42, 41], appears in the *Budget Justification*.

3 Literature Review

In addition to the above references, we list other works which have inspired this proposal.

Project 1: A hybrid manycore computing model for symbolic computation. Several papers present models for analyzing and improving the performances (w.r.t. memory access [19] or hardware resource management [47]) of the GPU implementation of a given algorithm. As discussed below, our first project brings performance analysis to a more abstract level as it aims at comparing algorithmic solutions for a given operation on an abstract hybrid manycore machine. Moreover, as it takes into account the heterogeneity of GPU-multicore architectures, our model is also closer to today's mainstream computer hardware than the PRAM model [29, 18] and systolic array model [48]. Since our model deals with algorithm analysis, it is more abstract than the OpenCL [49] programming model.

Project 2: Solving polynomial systems on GPUs. A few solvers based on numerical methods have been implemented on GPUs: see [51] for linear systems and [34] for non-linear ones. Numerical and

symbolic methods share many ideas: GPU implementation techniques of Fast Fourier Transforms, like the Stockham Algorithm, are an example; see [36] for the numerical case and [37] for the symbolic one. Numerical and symbolic approaches have also their specific challenges: numerical instability and intermediate expression swell, respectively. Symbolic solvers address this latter issue by means of the so-called *modular methods*, which reduce computations with bignum arithmetic to computations over finite fields, thus in fixed precision. Our paper [14] subscribes to this approach and we rely on it for this project. Symbolic solvers may have different output specifications; we use *triangular decompositions*, which are built on the mathematical notion of a *regular chain* [28]. Regular chains have attractive properties. In particular, each regular chain is a compact encoding of the solutions it represents; moreover its size can be sharply estimated [13], which is crucial in the design of efficient modular methods. Finally, our algorithm [11] computes triangular decompositions incrementally, that is, by solving one input equation after another. This feature plays a key role as explained below.

Project 3: Program analysis. The *polyhedron model* [16, 27, 3, 7, 2] is a powerful geometrical tool for analyzing the relation (w.r.t. data locality or parallelization) of the iterations of nested loop programs. Let us consider the case of parallelization. Once the polyhedron representing the *iteration space* of a loop nest is calculated, techniques of linear algebra and linear programming, can transform it into another polyhedron encoding the loop steps in a coordinate system based on time and space (processors). From there, a parallel program can be generated. To be practically efficient, one should avoid a too fine-grained parallelization; this is achieved by grouping loop steps into so-called *tiles*, which are generally trapezoids [26]. It is also desirable for the generated code to depend on parameters such as number of processors, cache sizes, etc. These extensions lead, however, to the manipulation of system of non-linear polynomial equations and the use of techniques like quantifier elimination, see [4]. Our project aims at advancing these techniques and the extensions of the *polyhedron model*.

4 Methodology

For each of the projects, we highlight a few techniques which are representative of this proposal.

Project 1: A hybrid manycore computing model for symbolic computation. In order to support GPU-multicore architectures, we consider an abstract machine with two types of processors: streaming multiprocessors (SMPs) and fast single-thread cores (FSTCs), both unlimited in numbers and all sharing an unlimited global memory. Each SMP can execute one SIMD (Single instruction, multiple data) program, with a fixed number of threads. Each FSTC has a cache memory and each SMP has a local memory, both types of memory having a finite size. We modify the fork-join multithreaded parallelism model [6] as follows. In the directed acyclic graph (DAG) representing the execution of a multithreaded program, we allow each node to be either a sequence of serial instructions running on one FSTC or a kernel call running on the SMPs. All thread blocks of a given kernel call start simultaneously and run concurrently on the SMPs. Moreover, data transfer between global memory and local memories occur only at the beginning and at the end of the execution of a thread block. Work and span are defined similarly to the fork-join parallelism model. Extending the ideas of [25] we call *burden* (overhead) a migration cost either due to a `fork` statement or due to a kernel call. Then, we define the *burdened span* as in [25], that is, the maximum length (burden) of a path in the burdened DAG.

We described a preliminary version of this model in [23] and applied it to the Euclidean Algorithm for computing polynomial greatest common divisors (GCDs). In this model, the GCD of two polynomials of degree n can be computed in time $O(n)$, with work $O(n^2)$, as in the systolic array model [8].

In addition, our implementation minimizes the burdened span such that we verified this linear time experimentally, for $n \leq 10,000$. Of course, this latter value is hardware dependent.

Several enhancements of this model are planned. For instance integrating cache memory considerations, thus extending the work of [17]. Another goal of this project is to revisit the fundamental algorithms in computer algebra [50] within this manycore computing model. The ultimate goal is to optimize those algorithms for implementation on manycore architectures. One way to do this is via *data reshaping*, see [44], that is, by mapping the input data set to another data set depending on a parameter, say s, and then choosing s such that it optimizes a complexity measure, like burdened parallelism. Clearly this goal is related to the component on automatic parallelization of Project 3.

Project 2: Solving polynomial systems on GPUs. As mentioned before, the challenge of this project is to select a solving algorithm which is suitable for GPU implementation. Ideally, the execution of such an algorithm should decompose into a series of steps, each of which to be efficiently performed on the device. To this end, each step must have sufficient work and offer opportunities for an SIMD parallelization. One should also be able to sharply estimate the minimum amount of memory which is necessary to perform that step. Indeed, the memory space required by a kernel call should be allocated before this kernel starts to execute. We argue in the following that our incremental algorithm [11] has the desired properties. If $f_1 = 0, \ldots, f_m = 0$ are the input equations and if, for some $1 \leq i < m$, the system consisting of the equations $f_1 = 0, \ldots, f_i = 0$ has already been solved producing output solution components $C_1, \ldots, C_e$, then the incremented system $f_1 = 0, \ldots, f_i = 0, f_{i+1} = 0$ is solved by applying a procedure called Intersect to each pair $(f_{i+1}, C_1), \ldots, (f_{i+1}, C_e)$. Generically, if the polynomials $f_1, \ldots, f_m$ have n variables, then performing the call Intersect(f_{i+1}, C_j) is done by executing at most $n - 1$ times a procedure which is essentially that of our GPU-based bivariate solver [38]. Therefore, we claim that our algorithm [11] is suitable for a GPU implementation, when coefficients are in a finite field. As mentioned, we will follow the modular method of [14]. Thus, other operations must receive GPU support in order to obtain the real solutions of an input system with rational number coefficients. These operations are Hensel lifting (for regular chains) and real root isolation, for which GPU-suitable algorithmic solutions exist.

Project 3: Program analysis. As noticed before, the use of techniques from polynomial algebra, in particular quantifier elimination (QE), is becoming an important research direction in automatic parallelization. The Authors of [4] observe, however, that classical algorithms for quantifier elimination are not suitable, since they do not always produce conjunctions of atomic formulas, while this format is required in order to generate code automatically. This issue is addressed by our recent algorithm for computing cylindrical algebraic decomposition [12]. Indeed, this algorithm supports QE in a way that the output of a QE problem has the form of a *case discussion*: this is appropriate for code generation.

5 Summary and Impact

We will design and deliver high-performance computing tools (model, algorithms, software packages) for solving polynomial systems symbolically on heterogeneous GPU-multicore architectures. We will apply these tools to program verification and automatic parallelization. The interaction between our core project and these applications will bring overall efficiency and robustness. I believe that the technology and software generated by this project will be major advances in symbolic computation. They will also provide high-performance scientific computing tools capable of tackling problems for which no software solutions exist today. The proposed research has clear potential for industrial applications.

References

[1] Parallel computing toolbox: Users guide. Technical Report R2012b, The MathWorks, Inc., 2012.

[2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proc. of the 19th joint European Conf. on Theory and Practice of Software, International Conf. on Compiler Construction* (CC/ETAPS), p. 244–263, Springer-Verlag, 2010.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. of PACT '04*, p. 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[4] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proc. of CC'10/ETAPS'10*, p. 283–303, 2010.

[5] E. Berberich, P. Emeliyanenko, and M. Sagraloff. An elimination method for solving bivariate polynomial systems: Eliminating the usual drawbacks. In *Proc. of ALENEX*, p. 35–47, 2011.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, June 2008.

[8] R. P. Brent, H. T. Kung, and F. T. Luk. Some linear-time algorithms for systolic arrays. In *Proc. of the IFIP Congress*, p. 865–876, 1983.

[9] C. Chen, J. H. Davenport, J. P. May, M. Moreno Maza, B. Xia, and R. Xiao. Triangular decomposition of semi-algebraic systems. In *Proc. of ISSAC*, p. 187–194, 2010.

[10] C. Chen and M. Moreno Maza. Semi-algebraic description of the equilibria of dynamical systems. In *Proc. of CASC*, p. 101–125, 2011.

[11] C. Chen and M. Moreno Maza. Algorithms for computing triangular decomposition of polynomial systems. *J. Symb. Comput.*, 47(6):610–642, 2012.

[12] C. Chen and M. Moreno Maza. An incremental algorithm for computing cylindrical algebraic decompositions. *Proc. of Asian Symposium of Computer Mathematics (ASCM) 2012*, Oct. 2012.

[13] X. Dahan, A. Kadri, and É. Schost. Bit-size estimates for triangular sets in positive dimension. *J. Complexity*, 28(1):109–135, 2012.

[14] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *Proc. of ISSAC*, p. 108–115, 2005.

[15] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, p. 133–137, New York, NY, USA, 2004. ACM.

[16] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[17] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory Comput. Syst.*, 45(2):203–233, 2009.

[18] P. B. Gibbons. A more practical PRAM model. In *Proc. of SPAA '89*, p. 158–168, 1989. ACM.

[19] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proc. of SC '06*, New York, NY, USA, 2006. ACM.

[20] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, p. 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[21] J. Grabmeier, E. Kaltofen, V. Weispfenning, eds. *Computer Algebra Handbook*, Springer, 2003.

[22] S. Haque and M. Moreno Maza. Determinant computation on the gpu using the condensation method. *J. of Physics: Conference Series*, 341, 2011.

[23] S. Haque and M. Moreno Maza. Plain polynomial arithmetic on gpu. *J. of Physics: Conference Series*, 385, 2012.

[24] S. Haque, S.Hossain, and M. Moreno Maza. Cache friendly sparse matrix-vector multiplication. In *Proc. of PASCO*, p. 175–176, New York, NY, USA, 2010. ACM.

[25] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *Proc. of SPAA*, p. 145–156, 2010.

[26] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Proc. of PoPL*, p. 160–173, New York, NY, USA, 1997. ACM.
[27] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. of PoPL*, p. 319–329, New York, NY, USA, 1988. ACM.
[28] M. Kalkbrener. A generalized Euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143–167, 1993.
[29] H. T. Kung. Why systolic architectures? *J. of IEEE Computer*, 15(1):37–46, 1982.
[30] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proc. of SC '01*, p. 55–55, New York, NY, USA, 2001. ACM.
[31] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Efficient evaluation of large polynomials. In *Proc. of ICMS*, p. 342–353, 2010.
[32] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *Proc. of ISSAC*, p. 239–246, New York, NY, USA, ACM, 2009.
[33] X. Li, M. Moreno Maza, R. Rasheed, and E. Schost. The `modpn` library: Bringing fast polynomial arithmetic into Maple. *J. Symb. Comput.*, 46(7):841–858, July 2011.
[34] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. A scalable high performance Cholesky factorization for multicore with GPU accelerators. In *Proc. of VECPAR*, p. 93–101, Berlin, Heidelberg, 2011. Springer-Verlag.
[35] L. Meng, Y. Voronenko, J. R. Johnson, M. Moreno Maza, F. Franchetti, and Y. Xie. Spiral-generated modular FFT algorithms. In *Proc. of PASCO*, p. 169–170, 2010. ACM.
[36] K. Moreland and E. Angel. The FFT on a GPU. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, p. 112–119, 2003.
[37] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a gpu. *J. of Physics: Conference Series*, 256, 2010.
[38] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a gpu. *J. of Physics: Conference Series*, 341, 2011.
[39] M. Moreno Maza and J.-L. Roch, editors. *PASCO '10: Proc. of the 4th International Workshop on Parallel and Symbolic Computation*, New York, NY, USA, 2010. ACM.
[40] M. Moreno Maza and S. Watt, editors. *PASCO '07: Proc. of the 2007 International Workshop on Parallel Symbolic Computation*, New York, NY, USA, 2007. ACM.
[41] M. Moreno Maza and R. Xiao. Generating program invariants via interpolation. *CoRR*, abs/1201.5086, 2012.
[42] M. Moreno Maza and R. Xiao. Degree and dimension estimates for invariant ideals of p-solvable recurrences. *Proc. of ASCM*, 2012.
[43] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multi-cores. In *Proc. of HPCS*, p. 378–399, Berlin, Heidelberg, 2010. Springer-Verlag.
[44] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.
[45] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.
[46] 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. *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.
[47] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. PPoPP '08*, p. 73–82, New York, NY, USA, 2008. ACM.
[48] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM J. Comput.*, 13(2):409–422, 1984.
[49] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2010.
[50] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 2003.
[51] F. Wei, J. Feng, and H. Lin. GPU-based parallel solver via the Kantorovich theorem for the nonlinear Bernstein polynomial systems. *Comput. Math. Appl.*, 62(6):2506–2517, Sept. 2011.