# Accelerating Polynomial Homotopy Continuation on Graphics Processing Units

by

Xiangcheng Yu
B.A. (Dalian University of Technology) 2010
M.S. (University of Illinois at Chicago) 2012

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois at Chicago, 2015

Chicago, Illinois

Defense Committee:
Jan Verschelde, Chair and Advisor
Rafail Abramov
David Nicholls
Lev Reyzin
Sonja Petrovic (Illinois Institute of Technology)

## ACKNOWLEDGMENTS

I am heartily thankful to my supervisor, Jan Verschelde, whose encouragement, guidance and support from the initial to the final level enabled me to study the subject, explore new technologies and develop our software packages.

It gives me great pleasure in acknowledging, Rafail Abramov, David Nicholls, Lev Reyzin, Sonja Petrovic, who served on the thesis defense committee. Thank you for your valuable comments and suggestions.

I would like to thank NSF for supporting my research in this thesis, under Grant No. 1440534 and Grant No. 1115777. I also would like to thank LAS-UIC for the fund of the computer workstation used through my research.

## TABLE OF CONTENTS

# TABLE OF CONTENTS (Continued)

# LIST OF TABLES

## LIST OF FIGURES

## LIST OF FIGURES (Continued)

**FIGURE** **PAGE**

## LIST OF ABBREVIATIONS

| | |
|---|---|
| CPU | Central Processing Unit, also called host in GPU computing |
| GPU | Graphics Processing Unit, also called device in GPU computing |
| CUDA | Compute Unified Device Architecture |
| CGI | Common Gateway Interface |
| GNU | Not Unix |
| GPL | Generic Public License |
| HTTPS | HyperText Transfer Protocol with Secure Sockets Layer |
| MGS | Modified Gram-Schmidt |
| PED | Polynomial Evaluation and Differentiation |
| PCI | Peripheral Component Interconnect |
| PHC | Polynomial Homotopy Continuation |
| RAM | Random-Access Memory |
| SDK | Software Development Kit |
| SHA | Secure Hash Algorithm |
| SIMD | Single Instruction, Multiple Data |
| SIMT | Single Instruction, Multiple Threads |
| SM | Streaming Multiprocessor |
| SQL | Structured Query Language |
| TCP | Transmission Control Protocol |

# SUMMARY

Polynomial homotopy continuation is a symbolic-numerical method to compute all solutions of a polynomial system. In recent years, Graphics Processing Units (GPU) offer much more computing power than Central Processing Units (CPU). There are many strong demands for parallel computing of polynomial homotopy continuation on GPU accelerators. Also, larger dimension systems and higher degrees are likely to have worse numerical conditions, so we expect to calculate with double double and quad double arithmetic to improve the quality.

In this thesis, an accelerated homotopy continuation method is designed on GPUs and achieves good speedups in multiple precisions. In the first chapter, we implement polynomial evaluation and differentiation of benchmark problems. A new tree mode is introduced for monomial evaluation to reduce global memory access. In the second chapter, we design Newton's method on GPUs, which minimizes the communication between the CPU host and the GPU device. In the third and fourth chapter, predictor-corrector algorithms are developed to track single path and multiple paths.

For another contribution, a web interface is designed to solve polynomial systems in the cloud. We want to classify polynomial systems and identify the polynomial systems we solved. For this problem, we represent polynomial systems by a new type of graph. Via the canonical form of this graph, a database is implemented for storing and searching polynomial systems.

# CHAPTER 1

# INTRODUCTION

Polynomial systems arise in many fields of science and engineering, like design of mechanisms, equilibia of chemical reactions, Nash equilibia, etc. Polynomial homotopy continuation is a symbolic-numerical method to compute all solutions of a polynomial system. As the number of solutions can grow exponentially in the degrees, the number of variables and equations, the computational complexity of these problems is hard. Also, as the degrees and the number of solutions increase, the numerical conditioning is likely to worsen as well. To improve the quality, we calculate with double double and quad double arithmetic.

In recent years, Graphics Processing Unit (GPU) accelerators have achieved exponential growth in both computing ability and memory bandwidth. Therefore, there is much interest in parallel computing of polynomial homotopy continuation on GPU accelerators. GPU accelerators provide a promising technology to deliver significant speedups over Central Processing Unit (CPU), but may require a complete overhaul of the algorithms in polynomial homotopy continuation.

In this thesis, a parallel implementation is developed for acceleration of polynomial homotopy continuation on GPUs (58; 59; 60), to obtain both speedup and quality up. The software package (57) is integrated into PHCpack (53) and phcpy (54). In addition, a web interface of PHCpack is created to grant users easy access to solve polynomial systems. Also, a polynomial database is built based on a canonical graph representation to classify polynomial systems (8).

## 1.1 Background

A polynomial $f(\mathbf{x})$ with $n$ unknowns $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined as

$$f(\mathbf{x}) = \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad \mathbf{a} = (a_1, a_2, \ldots, a_n), \quad c_{\mathbf{a}} \in \mathbb{C}, \quad c_{\mathbf{a}} \neq 0, \tag{1.1}$$

where $\mathbf{x}^{\mathbf{a}} = x_1^{\mathbf{a}_1} x_2^{\mathbf{a}_2} \cdots x_n^{\mathbf{a}_n}$. The finite set $A$ of exponent vectors $\mathbf{a}$ is called the support of the polynomial $f$. $c_{\mathbf{a}}$ is the nonzero coefficient for the monomial $\mathbf{x}^{\mathbf{a}}$. By default, the coefficients of the polynomials are complex numbers.

Given a polynomial system $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ of $N$ polynomials $\mathbf{f} = (f_1, f_2, \ldots, f_N)$, this thesis focuses on using homotopy continuation method to find all isolated solutions.

### 1.1.1 Polynomial homotopy continuation

To find all isolated solutions, we use polynomial homotopies (37; 49; 48). A homotopy connects the target system we want to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ to a start system $\mathbf{g}(\mathbf{x}) = \mathbf{0}$. Solutions of $\mathbf{g}(\mathbf{x})$ are easier to compute or given. A path from a known solution of $\mathbf{g}(\mathbf{x}) = \mathbf{0}$ to a solution of $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ is called a solution path.

A polynomial homotopy continuation method, as shown in Figure 1, consists of 5 stages:

1. We first construct a generic system $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, which has the same Newton polytopes as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, and use it in a homotopy, such as,

$$\mathbf{h}(\mathbf{x}, t) = \gamma(1-t)^k \mathbf{g}(\mathbf{x}) + t^k \mathbf{f}(\mathbf{x}) = \mathbf{0}, \quad t \in [0, 1], \tag{1.2}$$

Figure 1: A diagram of the polynomial homotopy continuation method.

$$\begin{tikzcd}
\mathbf{f}(\mathbf{x}) \arrow[r, "\text{1 parameter homotopy}", "\text{Newton polytopes}"'] & \mathbf{g}(\mathbf{x}) \arrow[r, "\text{2 polyhedral homotopy}", "\text{initial form}"'] & \mathbf{b}_i(\mathbf{x}) \arrow[d, "\text{3 solve}"', "\text{binomial}"] \\
\{\mathbf{x}\}_{\mathbf{f}(\mathbf{x})=\mathbf{0}} & \{\mathbf{x}\}_{\mathbf{g}(\mathbf{x})=\mathbf{0}} \arrow[l, "\text{4 path tracking}"', "\mathbf{h}(\mathbf{x},t)"] & \{\mathbf{x}\}_{\mathbf{b}_i(\mathbf{x})=\mathbf{0}} \arrow[l, "\text{5 path tracking}"', "\hat{\mathbf{g}}_i(\mathbf{x},t)"]
\end{tikzcd}$$

Note: It includes 5 stages from $\mathbf{f}(\mathbf{x})$ to its solution set $\{\mathbf{x}\}_{\mathbf{f}(\mathbf{x})=\mathbf{0}}$. Each stage is labelled by its index and method.

The coefficients $\mathbf{g}(\mathbf{x})$ and the constant $\gamma$ are random numbers to ensure the regularity of all solution paths. When $t = 0$, $\mathbf{h}(\mathbf{x}, 0)$ is the start system $\mathbf{g}(\mathbf{x})$, multiplied by $\gamma$. When $t = 1$, $\mathbf{h}(\mathbf{x}, 1) = \mathbf{f}(x)$, which is called the target system. Because a path might have more complicate numerical condition near the start point and the end point, we introduce a parameter $k = 2$ to increase the start range and the end range. For example, when $t = 0.1$, the position on the path $t^2$ is 0.01.

2. To solve $\mathbf{g}(\mathbf{x}) = \mathbf{0}$, we construct polyhedral homotopies (32) from a group of binomial systems $\mathbf{b}_i(\mathbf{x})$ to $\mathbf{g}(\mathbf{x})$, such as,

$$\hat{\mathbf{g}}_i(\mathbf{x}, t) = \sum_{\mathbf{a}\in A_i} \bar{c}_{i\mathbf{a}} t^{\theta_i(\mathbf{a})} \mathbf{x}^{\mathbf{a}}, \tag{1.3}$$

where $\theta_i(\mathbf{a})$ are generated decimal exponents and 2 of them are 0's for each equation. When $t = 0$, $\hat{\mathbf{g}}_i(\mathbf{x}, 0)$ is a binomial system $\mathbf{b}_i(\mathbf{x})$. When $t = 1$, $\hat{\mathbf{g}}_i(\mathbf{x}, 1) = \mathbf{g}(\mathbf{x})$. These binomial systems are constructed by initial forms of $\mathbf{g}(\mathbf{x})$.

3. We solve these binomial systems $\mathbf{b}_i(\mathbf{x})$ and get their solution sets $\{\mathbf{x}\}_{\mathbf{b}_i(\mathbf{x})=\mathbf{0}}$.

4. From the solutions of $\mathbf{b}_i(\mathbf{x})$ to the solutions of $\mathbf{g}(\mathbf{x})$, we apply path tracking methods to approximate solution paths $\mathbf{x}(t)$ defined by $\hat{\mathbf{g}}_i(\mathbf{x}, t) = \mathbf{0}$. Path tracking methods are called predictor-corrector methods. See Figure 2.

5. Similarly, from the solutions of $\mathbf{g}(\mathbf{x})$ to the solutions of $\mathbf{f}(\mathbf{x})$, we apply path tracking methods to $\mathbf{h}(\mathbf{x}(t), t) = \mathbf{0}$.

Polynomial homotopy continuation is a symbolic-numerical method. The constructions of the homotopies are symbolic from step 1 to 3, while path tracking is numerical in steps 4 and 5. The numerical part is more computational intensive and takes most of the solving time (48).

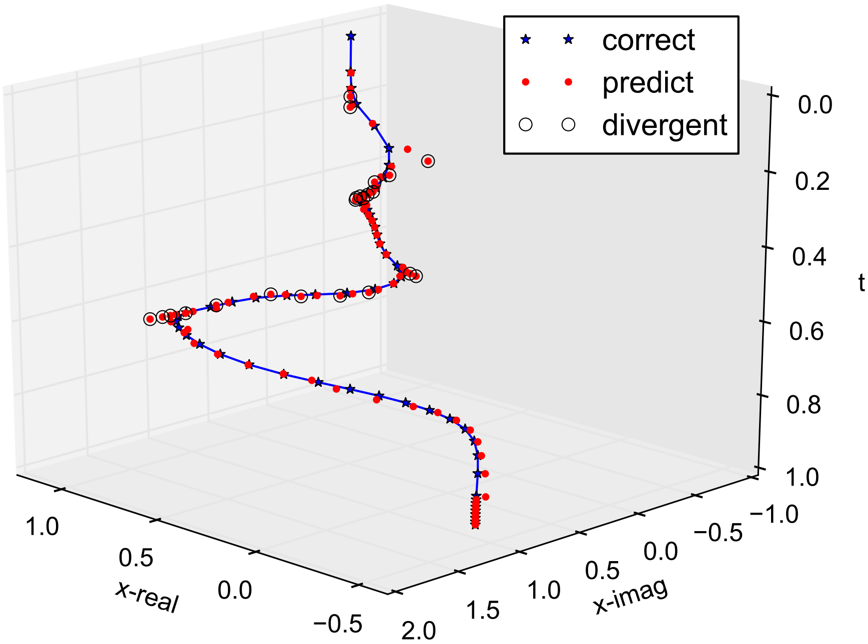
A solution path $\mathbf{x}(t)$ changes continuously as $t$ increases. When $t = 0$, it is the start solution. When $t = 1$, it is the target solution. From the start solution to the target solution, the predictor-corrector method is used to track the path numerically.

1. Single path tracking

   To track a single path, $t$ increases gradually each time by a small amount of the step size $\Delta t$. In each step, the predictor uses extrapolation of previous points to predict a point $\bar{\mathbf{x}}(t+\Delta t)$, which is close to the path solution $\mathbf{x}(t+\Delta t)$. Then the corrector uses Newton's method to get the local solution $\mathbf{x}(t + \Delta t)$.

   The step size $\Delta t$ is controlled by the correction result. If the correction fails, i.e. Newton's method does not converge, the step size $\Delta t$ is shortened. If corrections success consec-

Figure 2: Tracking one solution path in the view of one variable.



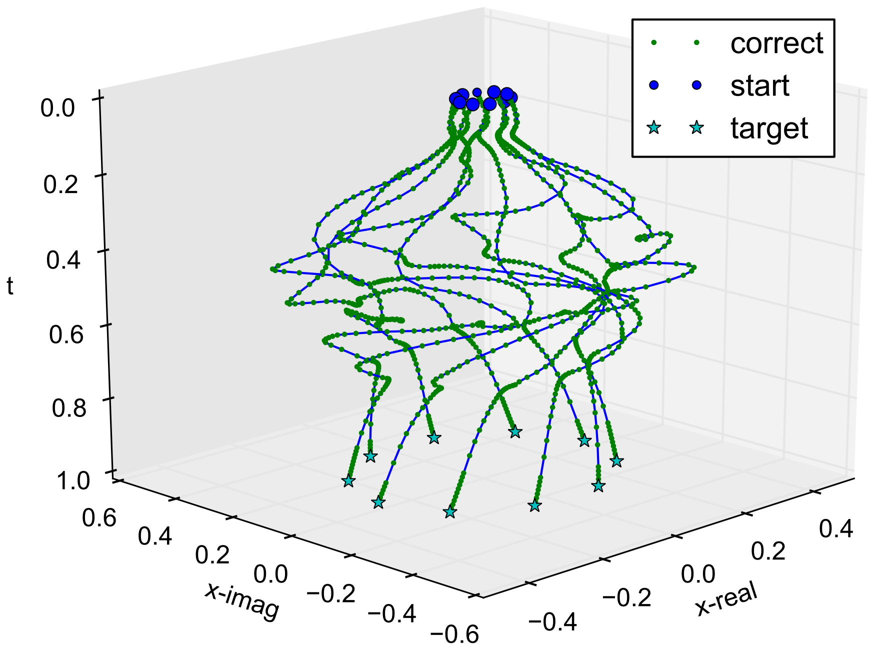

Note: Corrected points (marked by a star) are connected as they lie on the path. Points that are not connected are predicted points (marked by a dot). Predicted points from where the corrector diverged are marked by a circle.

utively for several times, the step size $\Delta t$ will be increased to make the path tracking faster. See Figure 2.

2. Muiltiple path tracking

In order to find all isolated solutions, we need to track multiple solution paths. Given a polynomial system with multiple start solutions $\mathbf{x}_i(0)$, all paths are tracked independently to the target solutions $\mathbf{x}_i(1)$. This is called multiple path tracking. See Figure 3.

Figure 3: Tracking multiple paths in the view of one variable.

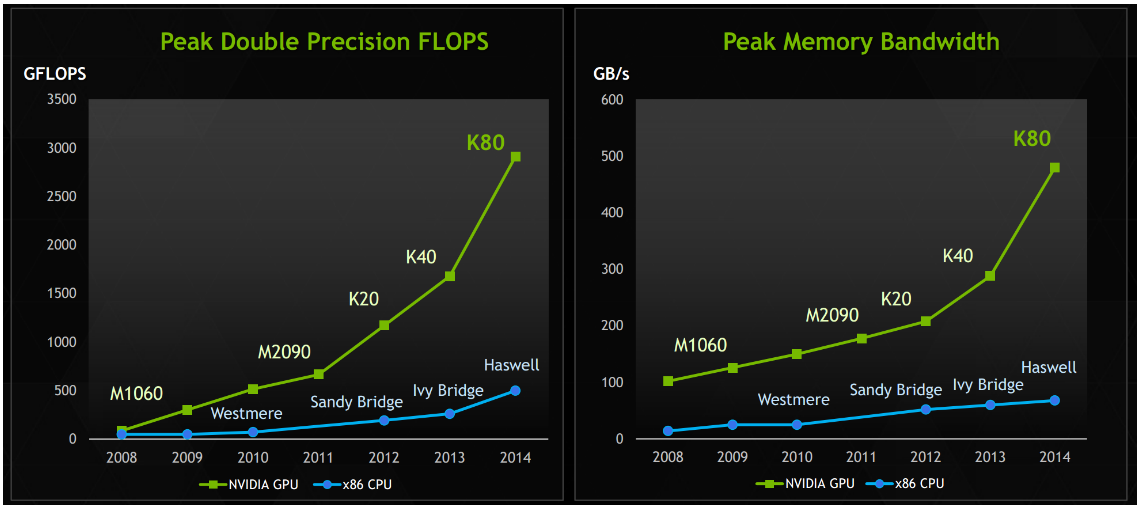



Note: Corrected points are marked by a smaller ball without boundary. Start solutions are marked by a bigger ball with boundary. Target solutions are marked by a star.

### 1.1.2 Graphics Processing Unit

In this thesis, we choose NVIDIA's CUDA(Compute Unified Device Architecture) as our Graphics Processing Unit (GPU) computing platform. In recent years, GPUs grow much faster than CPUs in terms of both peak computing performance and memory bandwidth. See Figure 4.

Figure 4: GPUs lead CPUs in peak double performance and memory bandwidth




Note: this chart is from the presentation (25)

The computational ability of the GPU comes from a large number of cores. For example, NVIDIA Tesla K20c has 2496 CUDA cores. These cores are grouped by Streaming Multiprocessors(SM). K20c has 13 SMs, and each SM has 192 CUDA cores.

For programming logical structure, a thread does a small job, a block has many threads working together, and a grid has many blocks working independently. Single instruction, multiple thread (SIMT) is a typical execution model on GPUs. A kernel for the GPU works like a function for the CPU. But a kernel launches for a grid, and it contains the same instructions for all threads in all blocks of this grid. Several kernels work sequentially to finish the entire problem. The CPU host controls the launches of these kernels and the sizes of their grids.

For GPU memory, there are three types, local memory, shared memory and global memory. Compared with CPU memory structure, local memory is like register, shared memory is like cache and global memory is like RAM. Modern CPU compilers can process these automatically, but GPU memory needs to be managed manually to obtain higher speedups.

Each type of GPU memory is related to the logical structure with limited life span. Local memory is used by the thread and disappears after the thread finished. Shared memory is used by the block, i.e. all threads in the same block, and disappears after the block finished. Global memory can be used by threads, blocks and grids, and disappears until the entire process is finished. Also, CPU host can read and write to global memory. See Figure 5.

Figure 5: GPU programming logical structure and memory access

| | Relationship with others | Cooperation with others | Memory |
|---|---|---|---|
| Thread | Parallel | Within the same block | Local, Shared, Global |
| Block | Parallel | No | Shared, Global |
| Grid | Sequential | By sequential kernels | Global |

From (28), in the CUDA execution model, there is a finer grouping of threads into warps. The warp size of all current CUDA-capable GPUs is 32 threads. Multiprocessors on the GPU execute instructions for each warp. From CUDA Toolkit Documentation (44), occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation. Occupancy is determined by three main factors: the among of registers (local memory) per thread, the size of shared memory per block and the number of threads per block. Occupancy decrease with the first two factors and increase with the third.

GPU memory should be managed carefully to obtain computation ability and memory bandwidth as much as possible. Here are some tips:

1. Local memory: each thread uses local memory as small as possible to increase occupancy.
2. Shared memory: each block uses shared memory as small as possible to increase occupancy. From (44), shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously. The warp size is 32 threads and the number of banks is also 32, so bank conflicts can occur between any threads in the warp. Ideally, all threads in a warp should read from different banks in the shared memory.
3. Global memory: from (28), the device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Each warp is executed in SIMD (Single Instruction, Multiple Data) fashion. Ideally, all threads in a warp should read from the consecutive part of memory together, to increase memory coalescing.

f

The computing ability and memory bandwidth are both the limits of GPUs' kernel performance. For K20c, memory bandwith is 208 GB/s, while its peak double float performance is 1.3 TFLOPS. If applications are memory-bound, we use the efficient memory bandwidth to measure our kernels. Another importance measurement in this thesis is speedups over one single CPU core.

Finally, the bandwidth between the CPU host and the GPU device is limited by PCI Express bus. For PCI Express 3.0, a 16-lane slot has only 15.754 GB/s, much less than GPU memory bandwidth. So we should minimize the amount of communication between the CPU host and the GPU device.

## 1.2 Problem Statement

Path tracking is a numerical computational intensive method, even more so in double double and quad double arithmetic. Tracking one single path is sequential. It might take hundreds of steps of predictions and corrections. Each correction, via Newton's method, costs several times of polynomial evaluation and differentiation (PED), and several times to solve linear system. The linear solver we choose is Modified Gram-Schmidt (MGS). To sum up, path tracking contains three major parts, Prediction, PED and MGS.

We combine these three computational intensive parts as the path tracker on GPUs. The challenge is to design massively parallel algorithms for this sequential problem and fit the GPU logical structure. The entire problem is split into several stages for grids. Then each stage is divided into independent jobs for blocks, and a job is divided into cooperative tasks for threads of a block.

With these three parts, the CPU host launches them in a dynamical sequence for unanticipated path convergent condition. The CPU host uses the previous results from GPU device to determine which is the next part. Also, the CPU host provides updated parameters to the GPU device. These communications between host and device should be minimized, due to the limited bandwidth.

## 1.3 Related work

Many software packages have been developed for polynomial homotopy continuation, e.g.: Bertini (7), HOM4PS (20), HOM4PS-2.0 (35), HOM4PS-3 (11), PHoM (24), NAG4M2 (36), HOMPACK (62; 63), PHCpack (53) and phcpy (54). Many of these packages are still under active development. To the best of our knowledge, our code provides the first path tracker for homotopy polynomial systems on GPUs.

To improve the quality, we calculate with double double and quad double arithmetic, using the QD library (29) on the CPU host and its CUDA version (40) on GPUs device.

For monomial evaluation and differentiation, we use reverse mode (23), originated in example of Speelpenning, to evaluate the derivative of monomials.

For GPU acceleration, the related work includes polynomial evaluation and differentiation (55; 64) and modified Gram-Schmidt (56; 64). In the first paper, polynomial evaluation and differentiation (PED) is implemented for randomly generated polynomial system. In the random polynomial system, each monomial has a fixed number of variables and each equation has a fixed number of monomials. Also, the dimension of polynomial system is 32 to fit warp size of GPU. In the second paper, modified Gram-Schmidt (MGS) is implemented for relatively small square matrix. Max dimension for complex double is 256, complex double double is 128 and complex quad double is 85.

Related research in computer algebra concerns the implementation of polynomial operations on GPUs. Reports on this research are (26) and (41). Computer algebra is geared towards exact computations, often over finite number fields. Our approach is numerical and we improve the accuracy of our results with double double and quad double arithmetic. This type of arithmetic is described in the section of error-free transformations in (46). Interval arithmetic on CUDA GPUs (14) is an alternative approach to improve the quality of numerical computations. Parallel automatic differentiation techniques on GPUs are described in (22). The computation of the Smith normal form as needed to solve large systems of binomials (that is: having exactly two monomials in every equation) using the NVIDIA GTX 780 graphics card is reported in (12) and in (13).

As for QR decomposition, many parallel implementations have been investigated by many authors, see e.g. (4), (5). In (10), the performance of CPU and GPU implementations of the Gram-Schmidt were compared. In (61), the left-looking scheme is dismissed because of its limited inherent parallelism and as in (61) we also prefer the right-looking algorithm for more thread-level parallelism. The application of extended precision to BLAS is described in (39), see (17) for least squares solutions. The implementation of BLAS routines on GPUs in triple precision (double + single float) is discussed in (43).

## 1.4 Contributions

The contributions of this thesis include three parts: accelerated polynomial homotopy continuation, a web interface of PHCpack and a polynomial database.

### 1.4.1 Accelerated polynomial homotopy continuation

The contributions of this thesis focus on accelerating polynomial homotopy continuation on GPUs for both single path (59) and multiple paths (60). In this process, Newton's method is implemented on GPUs (58). Also, previous work has been improved, including PED for real polynomial systems, Modified Gram-Schmidt (MGS) for large dimension matrix, and generalization of both PED and MGS for multiple paths. All these work are done in multiple precisions, including complex double, complex double double, complex quad double.

1. Accelerated polynomial homotopy continuation

   Accelerated polynomial homotopy continuation is designed and achieves good speedup. This is our major goal of all GPU implementations.

   In the single path tracking (59), we join the work of predictor, PDE and MGS together on GPUs. For predictor, we implement the Newton polynomial for different numbers of interpolations points. For corrector, we combine PED and MGS to Newton's method (58). The main challenge is that it is a sequential dynamic process, and thus the CPU host needs to control kernel launches according to the previous result from the GPU device. Each control process is designed with communication of only one double float.

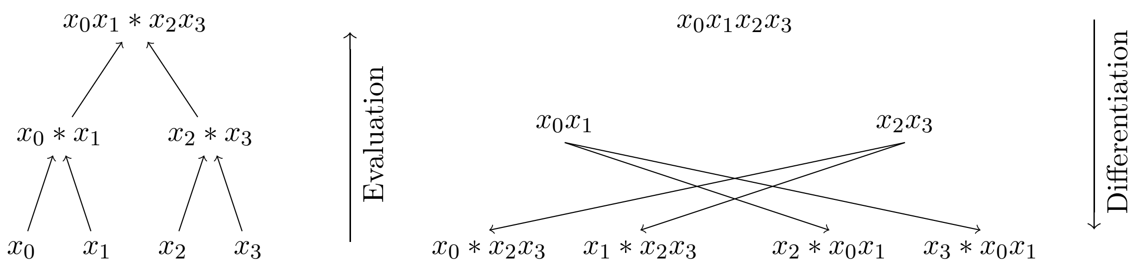
The multiple path tracking (60) generalizes the predictor-corrector method for thousands of independent paths. Because paths have different steps of predictor and corrector, we need to unify them into the same schedule of kernels. After each step, paths with new jobs are indexed. Predictor and corrector kernels use these indices to locate active jobs. This indexing is accelerated by applying GPU prefix sum. The CPU host needs only one integer from the GPU device, as the number of active jobs, to control kernel launches.

2. Improvement of Polynomial Evaluation and differentiation (PED)

   PED is improved for real polynomial systems and the dimension can go up to hundreds bounded by GPU's global memory. The relative work in (55) only works for randomly generated polynomial systems of dimension 32.

   For monomial evaluation and differentiation, a parallel tree mode (58) is developed and multiple threads can cooperate to evaluate the same monomial. See Figure 6.

Figure 6: Evaluate and differentiate of a monomial $x_0x_1x_2x_3$ by tree mode




(a) Evaluate monomial bottom up

(b) Compute partial derivatives top down

Compared with the reverse mode in the previous work, the tree mode can reduce the amount of global memory access, and thus it works better for complex double, which is bounded by memory bandwidth. But the tree mode limits the computation ability of all threads. So for higher precision, which is bounded by computation ability, the reverse mode fit better. Also, the reverse mode is redesigned to align the memory of instruction and monomial workspace. This helps us to get more memory coalescing and achieve better speed-up.

Multiple PEDs (60) is developed so that multiple paths can be computed simultaneously following the same instruction. The data structure is reorganized vertically for all evaluations, so there is more memory coalescing in monomial and summation kernels. With many PEDs, even for a small system like cylic-10, the speedup is better than that of a single large dimension polynomial system.

3. Improvement of Modified Gram-Schmidt (MGS)

   The block style of MGS is implemented on GPUs for large dimension matrices. The reduction step of MGS costs most of computation. With block style, multiple normalized columns can be used to reduce multiple unnormalized columns, and we can store multiple columns into shared memory for faster access.

   Multiple MGS is also implemented as different versions for small and large matrix. The small matrix can fit into the shared memory and be computed within a single GPU block. The large matrix is stored in different workspaces of the global memory. Multiple GPU blocks locate its own matrix workspace by the 3rd dimension of grid.

4. Software library

    The free and open source library (57) is developed to track a single path or many paths defined by a polynomial homotopy on GPUs. Built on NVIDIA graphics cards with CUDA SDKs, our code is released under the GNU GPL license.

    The main program that launches the accelerated path trackers starts with the definition of the polynomial homotopy and initializes the solution(s) at the start of the path(s). With PHClib, the C interface to PHCpack, the start system can be generated with its start solutions. Also, PHCpack provides condition number estimators at multiple precisions. These estimators can be applied to determine the precision required to reach a prescribed accuracy. Besides, via PHClib, we can call our GPU library from Python.

    Benchmarks on cyclic n-roots (52), Pieri (50; 38; 31), Nash (16; 42) indicate good speedup and quality up.

### 1.4.2 PHC web interface

The high speed internet and various types of user devices, like tablets and phones, inspire us to create cloud computing service for PHCpack. The advantages of PHC web interface for users include:

1. No software installation is required for the user.
2. Faster computation is hosted by our computational workstation.
3. Any device from computers, cell phones to tablets has the unified account access.
4. Easy graphic user interface helps the user to solve and manage polynomial systems.

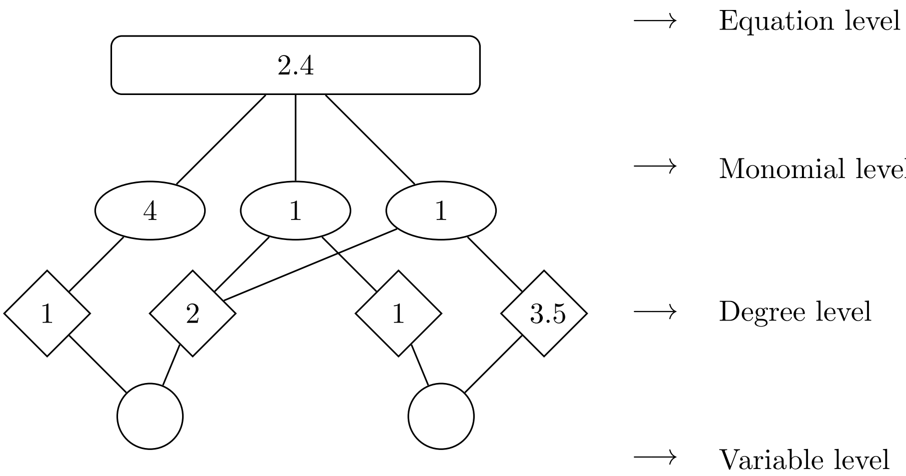
Its current version exports the blackbox solver (phc -b) and path tracker for a homotopy in one parameter (phc -p).

### 1.4.3 Polynomial database and search engine

A polynomial database is built based on a new type of polynomial graph. Each polynomial system is represented by a unique graph and a unique string. No matter the order of monomials, equations or different variable names, a polynomial systems always has a unique representation. See Figure 7.

Via the canonical form of this graph, a database and search engine of polynomial systems enables users to search by keywords of variable, monomial, equation or exact system.

Figure 7: The graph representation for a polynomial equation, $2.4 + 4x_1 + x_2x_1^2 + x_1^2x_2^{3.5}$.




Note: Type of nodes are represented as different shape: circle is for variable, diamond is for degree, ellipse is for monomial, rectangle is for equation.

## 1.5 Organization of this thesis

The major portion of this thesis is the accelerated polynomial homotopy continuation. We start from Chapter 2 for polynomial evaluation and differentiation(PED). In Chapter 3, we combine PED and modified Gram-Schmidt (MGS) as Newton's method. In Chapter 4, we develop the predictor and use Newton's method as the corrector to accelerate the single path tracking. Then we extend our implementation for the multiple path tracking in Chapter 5.

In Chapter 6, we develop a web interface of PHCpack for users to solve polynomial system in the cloud. In Chapter 7, we present a graph representation for polynomial systems, and then use its canonical form to design a database for polynomial systems.

# CHAPTER 2

# POLYNOMIAL EVALUATION AND DIFFERENTIATION ON GPUS

In this chapter, parallel algorithms are given for polynomial evaluation and differentiation (PED) on GPUs. Homotopy polynomial systems are studied as special cases. The problem is split into three parts: the evaluation of homotopy coefficients, the evaluation and differentiation of monomials, and the summation to the Jacobian matrix. For monomial evaluation, a new tree mode is developed and compared with reverse mode. For multiple evaluations of a polynomial system, more memory coalescing is achieved by transposing memory structure of multiple workspace, which leads to even better speedup. In this chapter, we group all ideas specific for PED in (58; 59; 60).

## 2.1 Overview

A homotopy polynomial system, such as (Equation 1.2) or (Equation 1.3), is a special case of the polynomial system like (Equation 1.1). A homotopy polynomial system has a starting system $\mathbf{f}(\mathbf{x})$ and a target system $\mathbf{g}(\mathbf{x})$, which share similar supports. To save computation, we join the starting system and the target system into one polynomial system, whose coefficients changes with respects of $t$.

$$h_i(\mathbf{x}) = \sum_{a \in A_{f_i} \cup A_{g_i}} p_{\mathbf{a}}(c_{\mathbf{a}}^{(f_i)}, c_{\mathbf{a}}^{(g_i)}, t)\mathbf{x}^{\mathbf{a}}, \tag{2.1}$$

where $c_{\mathbf{a}}^{f_i}$ is the coefficient for the monomial $\mathbf{x}^{\mathbf{a}}$ in $f_i$. $c_{\mathbf{a}}^{g_i}$ is the coefficient for the monomial $\mathbf{x}^{\mathbf{a}}$ in $g_i$. A finite set $A_{f_i}$ is the support of a polynomial $f_i$. A finite set $A_{g_i}$ is the support of a polynomial $g_i$. Meanwhile, $p_{\mathbf{a}}$ is a function for homotopy parameters.

A polynomial system is a set of polynomials, and a polynomial is a set of monomials. On CPUs, monomials of polynomials are handled sequentially. For each monomial, we follow three steps: compute coefficient, evaluate the monomial and its derivatives, and then add these values to the polynomials and to Jacobian matrix. On GPUs, these three steps are joint for all monomials in all polynomials. In this way, threads on GPUs work in parallel under the same instruction for each step. See Figure 8.

Figure 8: Compare pseudo code of the host and the device for PED.

*Pseudo code on the CPU host:*
for each polynomial do
  for each monomial do
    1. compute the coefficient $c(t)$ for this monomial;
    2. evaluate the monomial and its derivative;
    3. add the values to the polynomials and to the Jacobian matrix.

*Pseudo code on the GPU device:*
1. compute the coefficient $c(t)$
   for all monomials in all polynomials;
2. evaluate the monomial and its derivatives
   for all monomials in all polynomials;
3. add to the value of the polynomial and to the Jacobian matrix
   for all monomials in all polynomials.

In the process of computing a homotopy, there are a lot of variables and parameters. To manage memory more easily, we group all elements into instructions($Inst$), workspaces($W$) and parameters($P$). $Inst$ includes all static instructions, like coefficients of the start and target systems, positions and degrees of monomials, summation index, etc. $W$ includes all intermediate and final results, like variable's value $x$, coefficients of the homotopy system, values and derivatives of monomials, Jacobian matrix, etc. $P$ includes all parameters for Newton's method and path tracking, which are discussed in the next chapters.

Algorithm 1 indicates the upper level structure of PED. The details of the algorithm are discussed in the following sections.

**Algorithm 1** Polynomial evaluation and differentiation on GPU

1: **procedure** GPU_PED($Inst, W$)
2: launch kernel(s) GPU_PED_COEF($Inst.coef$, $W.coef$)
3: launch kernel(s) GPU_PED_MON($Inst.mon$, $W.x$, $W.mon$)
4: launch kernel(s) GPU_PED_SUM($Inst.sum$, $W.mon$, $W.matrix$)
5: **end procedure**

## 2.2 Monomial evaluation and differentiation

For a monomial with $k$ variable $c_{\mathbf{a}} x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}}$, where $a_{i_k} \neq 0$, its derivatives are

$$c_{\mathbf{a}} a_{i_1} x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}},\ c_{\mathbf{a}} a_{i_2} x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}},\ \ldots,\ c_{\mathbf{a}} a_{i_k} x_{i_1}^{a_{i_1}} x_{i_2}^{a_{i_2}} \cdots x_{i_k}^{a_{i_k}-1} \tag{2.2}$$

These derivatives and monomial value share the common factor $c_{\mathbf{a}} x_{i_1}^{a_{i_1}-1} x_{i_2}^{a_{i_2}-1} \cdots x_{i_k}^{a_{i_k}-1}$. Thus, the common factor can be pre-computed with its coefficient. Also, exponents can be multiplied to derivatives independently. Without common factor and exponents, the derivatives are

$$x_{i_2} x_{i_3} \cdots x_{i_k},\ x_{i_1} x_{i_3} \cdots x_{i_k},\ \ldots,\ x_{i_1} x_{i_2} \cdots x_{i_{k-1}} \tag{2.3}$$

This series of product is called Speelpenning. Instead of computing each of them with $O(k^2)$ multiplications, we can use reverse mode to compute them with $O(k)$ multiplications.

This section discusses two modes to evaluate Speelpenning, the reverse mode and a new tree mode. The reverse mode is a sequential algorithm for a monomial and each thread evaluates one monomial. To use reverse mode efficiently on GPUs, we introduce a warp-aligned data structure. A new tree mode is developed to evaluate one monomial by multiple threads. The comparison of these two modes shows that the tree mode works better for double, and the reverse mode works better for double double and quad double.

### 2.2.1 Reverse mode

The reverse mode (23) contains three parts, forward product, backward product and cross product. See Figure 9.

For reverse mode on GPUs, each thread evaluates one monomial. The intermediate results of the forward product need to be stored. But the backward product and cross products are combined, and the final results can overwrite that of forward product. To provide similar

Figure 9: Reverse mode to compute the value and derivatives of monomial $x_0x_1x_2x_3$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Forward | product | $\rightarrow$ | | $x_0$ | $x_0 * x_1$ | $x_0x_1 * x_2$ | $x_0x_1x_2 * x_3$ |
| Backward | product | $\leftarrow$ | $x_1 * x_2x_3$ | $x_2 * x_3$ | $x_3$ | | |
| Cross | product | $\downarrow$ | $x_1x_2x_3$ | $x_0 * x_2x_3$ | $x_0x_1 * x_3$ | $x_0x_1x_2$ | $x_0x_1x_2 * x_3$ |

workload to all threads in one block, the monomials have been sorted by number of variables. See Figure 10.

For small monomials, the intermediate results of the forward product can be stored in the shared memory. For large monomials in polynomial system of high dimension, like cyclic-32, each thread needs 32 data positions to store the intermediate results, and the shared memory is not enough for all threads. Thus, we have to use global memory.

To use global memory efficiently, we want to create memory coalescing for all threads in a warp. The memory access in reverse mode contains two parts: position instructions, values and derivatives of monomials. For the monomial set in Figure 10, the position instruction table is in Figure 11.

The data structure of position instruction is improved for more memory coalescing. The original way is organized by monomials, i.e $n\_var_{tidx}$ plus position array. But it costs random memory access. To create memory coalescing, we can align position instructions for 32 threads in a warp, by recording the table of Figure 11 row by row. With some empty position elements, the size of the joint position array of a warp is $32 * \max(n\_var_{tidx})$. Plus $n\_var$ array, the total size of instruction array is $32 * (\max(n\_var_{tidx}) + 1)$. For polynomial system of high dimension,

Figure 10: Evaluating four monomials $x_0x_1x_2$, $x_3x_4x_5$, $x_2x_3x_4x_5$, $x_0x_1x_3x_4x_5$.

| tidx | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $m_{tidx}$ | $x_0x_1x_2$ | $x_3x_4x_5$ | $x_2x_3x_4x_5$ | $x_0x_1x_3x_4x_5$ |
| $\frac{\partial m_{tidx}}{\partial x_j}$ | $x_0$<br>$x_0 \star x_1$ | $x_3$<br>$x_3 \star x_4$ | $x_2$<br>$x_2 \star x_3$<br>$x_2x_3 \star x_4$ | $x_0$<br>$x_0 \star x_1$<br>$x_0x_1 \star x_3$<br>$x_0x_1x_3 \star x_4$ |

forward

(a) the forward product

| tidx | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $m_{tidx}$ | $x_0x_1x_2$ | $x_3x_4x_5$ | $x_2x_3x_4x_5$ | $x_0x_1x_3x_4x_5$ |
| $\frac{\partial m_{tidx}}{\partial x_j}$ | $x_1 \star x_2$<br>$x_0 \star x_2$<br>$x_0x_1$ | $x_3 \star x_4x_5$<br>$x_3 \star x_5$<br>$x_3x_4$ | $x_3 \star x_4x_5$<br>$x_2 \star (x_4 \star x_5)$<br>$x_2x_3 \star x_5$<br>$x_2x_3x_4$ | $x_1 \star x_3x_4x_5$<br>$x_0 \star (x_3 \star x_4x_5)$<br>$x_0x_1 \star (x_4 \star x_5)$<br>$x_0x_1x_3 \star (x_5)$<br>$x_0x_1x_3x_4$ |

back & cross

(b) the backward and cross products

Note: The tidx stands for the thread index.

after sorting monomials by number of variables, the *n_var*s of all threads in one warp tends to be similar. Thus, without adding too many empty position instructions, we can create memory coalescing for position reading.

Comparing Figure 10 and Figure 11, it is clear that the values of each monomial and its derivatives have similar structure as the position instruction of this monomial. The value corresponds to *n_var* and derivatives corresponds to *pos* array. Thus, we can adapt exact the same warp-aligned data structure to store monomials' values and derivatives, including forward results.

Figure 11: Position instruction of four monomials $x_0x_1x_2$, $x_3x_4x_5$, $x_2x_3x_4x_5$, $x_0x_1x_3x_4x_5$.

| tidx | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $n\_var_{tidx}$ | 3 | 3 | 4 | 5 |
| $pos_{tidx}$ | 0 | 3 | 2 | 0 |
| | 1 | 4 | 3 | 1 |
| | 2 | 5 | 4 | 3 |
| | | | 5 | 4 |
| | | | | 5 |

Note: The tidx stands for the thread index. For each $tidx$, $n\_var_{tidx}$ is the number of variables and $pos_{tidx}$ is the position array.

### 2.2.2 Tree mode

A new tree mode is discovered to evaluate Speelpenning of a monomial. Reverse mode is sequential for a single monomial, but tree mode enables several threads to evaluate the same monomial together.

Tree mode contains two steps:

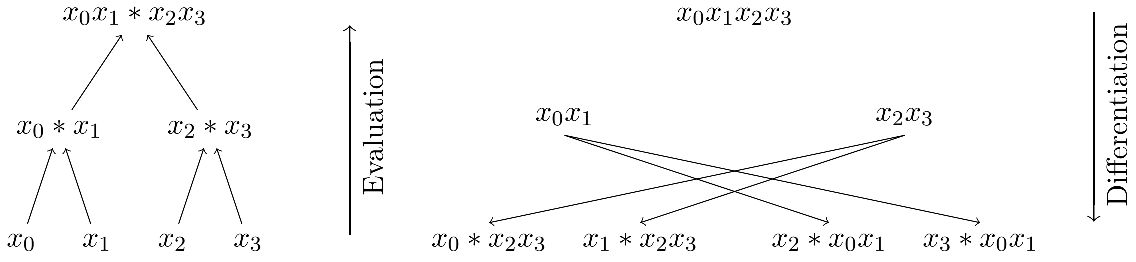
1. Evaluation monomial bottom up like parallel reduction. See Figure 12 (a)

2. Differentiation top down like cross product. See Figure 12 (b)

The mathematical logic of the second part differentiation is product rule of computing derivatives:

$$\frac{d}{dx}(u \cdot v) = \frac{du}{dx} \cdot v + u \cdot \frac{dv}{dx} \tag{2.4}$$

Figure 12: Evaluate and differentiate of a monomial $x_0x_1x_2x_3$ by tree mode




(a) Evaluate monomial bottom up

(b) Compute partial derivatives top down

If a monomial is splited into two parts $u$ and $v$, $u$ and $v$ have no common variable. Suppose $u = x_{i_1} \cdots x_{i_p}$ and $v = x_{j_1} \cdots x_{j_q}$,

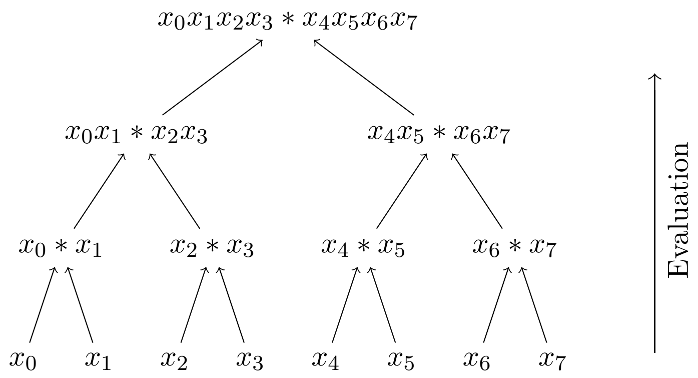
$$\frac{d}{dx_{i_*}}(u \cdot v) = \frac{du}{dx_{i_*}} \cdot v \qquad \frac{d}{dx_{j_*}}(u \cdot v) = u \cdot \frac{dv}{dx_{j_*}} \tag{2.5}$$

The derivatives of $u$ multiply $v$ and the derivatives of $v$ multiply $u$. This is the reason of that there is a cross product between each two terms on each level.

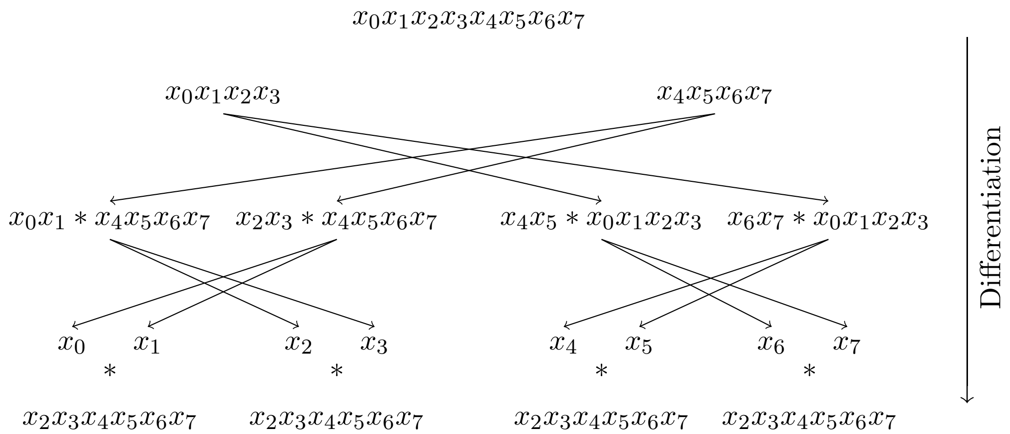
For GPU implementation of tree mode, the intermediate results of evaluation step are used by differentiation step, so we store them in shared memory. To optimize our kernel, we apply the techniques of parallel reduction (27), because parallel reduction has similar tree structure. Three strategies from parallel reduction are used:

1. Sequential addressing: it can solve shared memory back conflicts.

Figure 13: Evaluate and differentiate of a monomial $x_0x_1x_2x_3x_4x_5x_6x_7$ by tree mode




(a) Evaluate monomial bottom up




(b) Compute partial derivatives top down

2. unroll last warp: there is no need to __syncthreads(), because instruction are synchronous within a warp of 32 threads.

3. unroll completely: block size is fixed (512 for double), so "for" loop can be unrolled completely to save iteration instructions.

For double precision, tree mode have low arithmetic intensity and they are memory bounded. So we use memory bandwidth to measure the efficiency of our kernel. See Figure 14.

Figure 14: Evaluation and differentiation of 65,024 monomials in 1,024 doubles.

| | method | time | bandwidth | speedup |
|---|---|---|---|---|
| CPU | | 330.24ms | | |
| GPU | reverse mode | 86.43ms | | 3.82 |
| | tree mode naive | 15.54ms | 79.81GB/s | 21.25 |
| | sequential addressing | 14.08ms | 88.08GB/s | 23.45 |
| | unroll last warp | 10.19ms | 121.71GB/s | 32.40 |
| | unroll completely | 9.10ms | 136.28GB/s | 36.29 |

Note: Times on the K20C obtained with `nvprof` (the NVIDIA profiler) are in milliseconds (ms). Dividing the number of bytes read and written by the time gives the bandwidth. Times on the CPU are on one 2.6GHz Intel Xeon E5-2670, with code optimized with the `-O2` flag.

Tree mode enables several threads on GPUs to evaluate a monomial. But the number of threads in GPU block is typically $32n$. To adapt tree mode for monomials of any size, we can adjust the first level in tree mode. We can use $2^n$ threads evaluate for a monomial of size $2^n + 1$

**Algorithm 2** Single Monomial evaluation and differentiation by GPU block

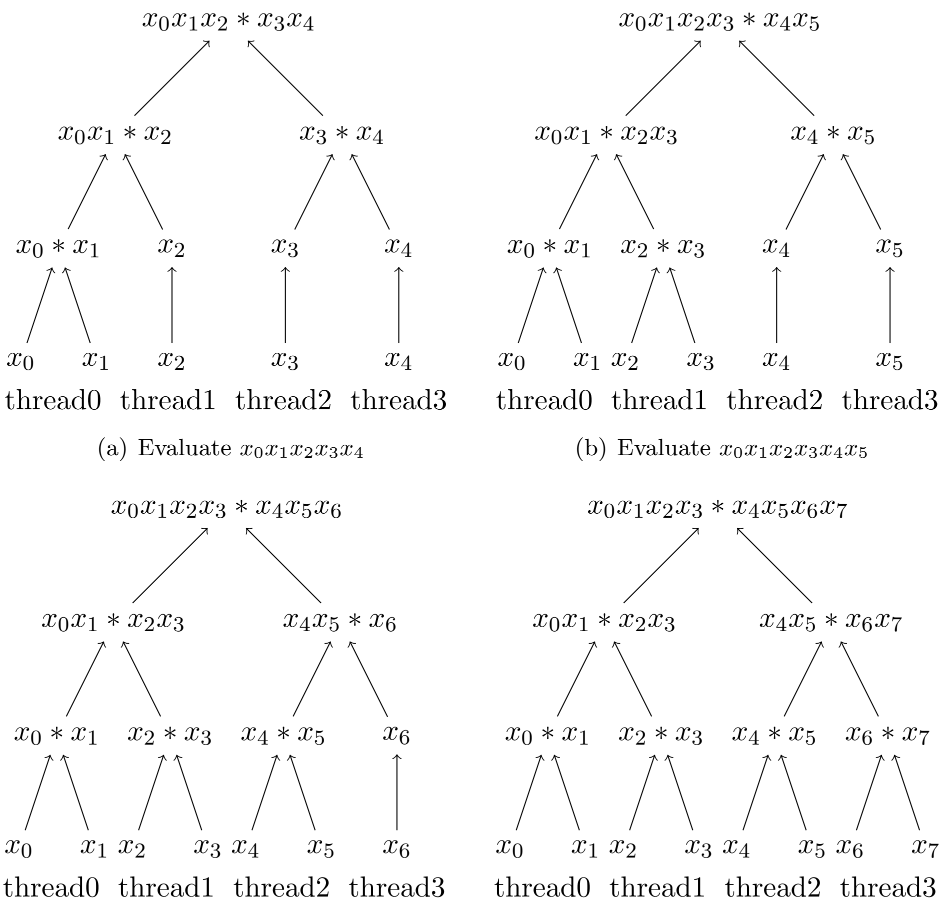
```
procedure GPU_Mon(X, n)
    load X into shared memory x
    nl ← n
    xlevel ← x
    for nl > 1 do
        xlast ← xlevel
        nl ← ⌈nl⌉
        if idx < nl then
            xlevel[idx] ← xlast[2 * idx] * xlast[2 * idx + 1]
        end if
        local barrier
    end for
    nl ← 2
    if idx = 0 then
        CommonFactor ← base * coef
    end if
    xlevel ← xlevel − nl
    if idx < 3 then
        xlevel[idx]* = CommonFactor
    end if
    while nl < n do
        xlast ← xlevel − 2 * nl
        if idx < nl then
            newidx ← idx XOR 1
            xlast[2 * idx]* = xlevel[newidx]
            xlast[2 * idx + 1]* = xlevel[newidx]
        end if
        nl ← 2 * nl
        xlevel ← xlast
        local barrier
    end while
end procedure
```

to $2^{n+1}$. To be specific, to evaluate a monomial of size $2^n + k$, where $0 < k \leq 2^n$, the first $k$ threads are in charge of $2k$ variables and the rest $2^n - k$ threads are in charge of $2^n - k$ variables. The first $k$ threads multiply two variables at the beginning, but the rest with one variable don't need to. See Figure 15.

Furthermore, the number of threads in GPU block is typically $32n$ and polynomial systems often have a lot of monomials of size smaller than 32. Thus, we want to use a block to evaluate multiple monomials simultaneously. For $2^n$ threads, it can evaluate a monomial of size $2^n + 1$. But if we stop evaluation before the top level, it can evaluate two monomials of size $2^n$. For example, 4 threads can evaluate a monomial of size 8, and they can evaluate 2 monomials together, too. See Figure 16.

For a block of 512 threads, we can evaluate 1 monomial of size 1024, 2 monomials of size 512, 4 monomials of size 256, etc. In the test of double precision, we can get speedup for all size of monomials. See Figure 17.

Figure 15: Evaluate monomials of size 5 to 8 by 4 threads in tree mode



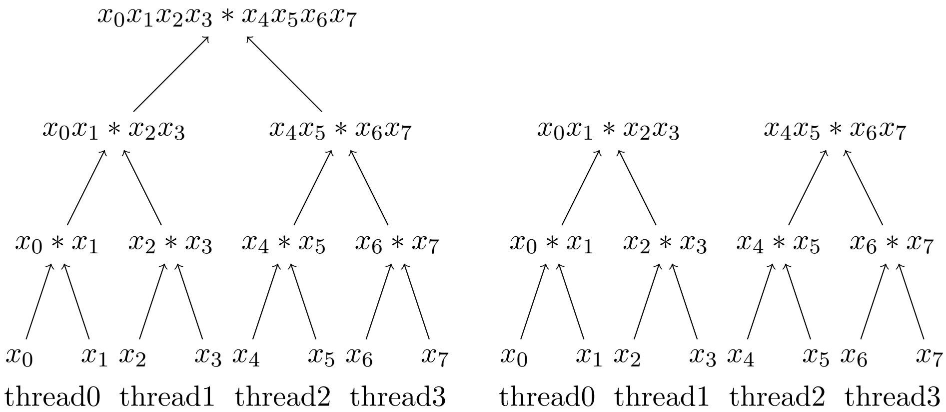

(a) Evaluate $x_0x_1x_2x_3x_4$

(b) Evaluate $x_0x_1x_2x_3x_4x_5$

(c) Evaluate $x_0x_1x_2x_3x_4x_5x_6$

(d) Evaluate $x_0x_1x_2x_3x_4x_5x_6x_7$

Figure 16: Evaluate single or multiple monomials in the same block by tree mode




(a) Evaluate $x_0x_1x_2x_3x_4x_5x_6x_7$ (b) Evaluate $x_0x_1x_2x_3$ and $x_4x_5x_6x_7$

### 2.2.3 Comparision of reverse mode and tree mode

In this subsection, reverse mode and tree mode are compared and tested for the evaluation of multiple monomials.

About complexity to evaluate a monomial of size $k$, reverse mode and tree mode have similar $3k+C$ multiplications. Reverse mode costs $k+C$ multiplication for forward product, backward product and cross product. Tree mode costs $\frac{1}{2}k + (\frac{1}{2})^2k + \cdots + 1 = k + C$ multiplication for evaluation bottom up, another $k + C$ for differentiation top down without last level, and on the last level, it takes $k$ multiplications for each variable. Thus tree mode costs $3k + C$ multiplication, too.

Figure 17: Evaluation and differentiation of $m$ monomials of different size $n$

| $n$ | $m$ | CPU | GPU | speedup |
|---|---|---|---|---|
| 1024 | 1 | 330.24ms | 9.12ms | 36.20 |
| 512 | 2 | 328.92ms | 8.73ms | 37.66 |
| 256 | 4 | 320.78ms | 8.84ms | 36.29 |
| 128 | 8 | 309.02ms | 8.15ms | 37.89 |
| 64 | 16 | 289.30ms | 7.27ms | 39.77 |
| 32 | 32 | 256.07ms | 9.51ms | 26.94 |
| 16 | 64 | 230.34ms | 8.86ms | 25.99 |
| 8 | 128 | 218.74ms | 7.79ms | 28.07 |
| 4 | 256 | 202.20ms | 7.05ms | 28.69 |

Note: by 65,024 blocks with 512 threads per block for 1,024 doubles in shared memory, accelerated by the K20C with timings in milliseconds obtained by the NVIDIA profiler. Times on the CPU are on one 2.6GHz Intel Xeon E5-2670, with code optimized with the -O2 flag.

For implementations on GPUs, memory bandwidth and thread ability are two important factors. For reverse mode, each thread has its own monomial to compute and thread ability is not limited, but shared memory is not enough for the intermediate results of all threads, and intermediate results need to be stored in the global memory and costs more global memory access. On the other hand, reverse mode can use shared memory to store intermediate results and reduce global memory access. The disadvantage is that tree structure limits thread computation in upper levels. On $n^{\text{th}}$ level, only $(\frac{1}{2})^n$ of all threads are used.

To sum up, tree mode works better for lower precision like complex double, which is memory bounded. For complex double-double and complex quad-double, which are more computation intensive, reverse mode is more efficient. See Figure 18.

Figure 18: Compare reverse mode and tree mode for a $k$-variable monomial

| | Reverse mode | Tree mode |
|---|---|---|
| Multiplications | $3k + C$ | $3k + C$ |
| Global memory | full access | half access |
| Shared memory | None | k |
| Thread ability | no limitation | limited on upper levels |
| Usage | complex double double,<br>complex quad double | complex double |

### 2.3 Homotopy coefficient evaluations

For homotopy coefficient, we can evaluated in a single kernel. Preprocessing of coefficient can reduce total computations.

#### 2.3.1 Coefficient-parameter homotopy

$$\mathbf{h}(\mathbf{x}, t) = \gamma(1-t)^k \mathbf{g}(\mathbf{x}) + t^k \mathbf{f}(\mathbf{x}) = \mathbf{0}, t \in [0, 1] \tag{2.6}$$

$\mathbf{h}(\mathbf{x}, t)$ could be considered as a general polynomial function. But after expending $t$ terms, the number of monomials will be increased by $k$ times. For this special polynomial system, we can simplify this homotopy and save computation.

The parameters $\gamma(1-t)^k$ and $t^k$ are constants for all monomials. Also, there are many monomials with the same support in $\mathbf{f}$ and $\mathbf{g}$, because $\mathbf{g}_i$ shares the same Newton polytope with $\mathbf{f}_i$. Like the following example(Gaussian quadrature formula 4),

$$
\begin{array}{llll}
f_1 = w_1 + w_2 - 1 & g_1 = (-0.38 - 0.93I)w_1 & +(-0.06 + 0.99I)w_2 & +(-0.91 - 0.42I) \\
f_2 = w_1x_1 + w_2x_2 & g_2 = (-0.99 - 0.08I)w_1x_1 & +(-0.45 + 0.89I)w_2x_2 & \\
f_3 = w_1x_1^2 + w_2x_2^2 - 2 & g_3 = (0.82 + 0.58I)w_1x_1^2 & +(0.94 + 0.33I) & \\
f_4 = w_1x_1^3 + w_2x_2^3 & g_4 = (-0.48 + 0.88I)w_1x_1^3 & +(-0.81 + 0.59I)w_2x_2^3 &
\end{array}
$$

By simplification, each equation has the following form:

$$h_i = \sum_{a \in A_f \cup A_g} (\gamma(1-t)^k c_{\mathbf{a}}^{(f)} + t^k c_{\mathbf{a}}^{(g)}) \mathbf{x}^{\mathbf{a}} \tag{2.7}$$

where $c_{\mathbf{a}}^{(f)}$ is the coefficient of monomial $\mathbf{x}^{\mathbf{a}}$ in $f$ and $c_{\mathbf{a}}^{(g)}$ is that of $g$. $A_f$ is the support set of $f$ and $A_g$ is the support set of $g$. After preprocessing coefficients, the number of monomials of homotopy is the sum of all different monomials in $f$ and $g$. $\gamma(1-t)^k$ and $t^k$ can be computed first and then all coefficients can be evaluated in parallel.

### 2.3.2 Polyhedral homotopy

Give a group of polyhedral homotopies,

$$\hat{\mathbf{g}}_i(\mathbf{x}, t) = \sum_{\mathbf{a} \in A_i} \bar{c}_{i\mathbf{a}} t^{\theta_i(\mathbf{a})} \mathbf{x}^{\mathbf{a}} \tag{2.8}$$

where $\theta_i(\mathbf{a})$ are generated decimal exponent and 2 of them are 0s for each equation.

The coefficients of $\mathbf{x}^{\mathbf{a}}$ in $\hat{\mathbf{g}}_i(\mathbf{x}, t)$ are the same for all $i$'s. Thus, we can evaluates the same $a$ of all $i$'s together. $\theta_i(\mathbf{a})$ are stored vertically to align data of the same $a$, so for the same $a$, we can have more memory coalescing.

## 2.4 Multiple evaluations

To find all isolated solutions for a polynomial system, we need to track multiple solution paths. During this process, we need to compute multiple evaluations of the same polynomial system with different points of $\mathbf{x}(t)$'s.

We can use similar strategies of the single evaluation: evaluate coefficients, evaluate monomials and then add values of the monomial workspace to Jacobian matrix. In the last step of the single evaluation, the summation of Jacobian matrix need random global memory access, because the partial derivatives for the same variable are in different monomials. But for the strategy of multiple evaluations, we can organize the workspace vertically to avoid random global memory access.

Figure 19: Tranposition of multiple monomial workspaces at different points.

| | monomials in memory | | | |
|---|---|---|---|---|
| path 0 | $a_0a_1a_2$ | $a_1a_2$ | $a_0a_2$ | $a_1a_2$ |
| path 1 | $b_0b_1b_2$ | $b_1b_2$ | $b_0b_2$ | $b_1b_2$ |
| path 2 | $c_0c_1c_2$ | $c_1c_2$ | $c_0c_2$ | $c_1c_2$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |

(a) Multiple horizontal workspaces

| path 0 | path 1 | path 2 | $\cdots$ |
|---|---|---|---|
| $a_0a_1a_2$ | $b_0b_1b_2$ | $c_0c_1c_2$ | $\cdots$ |
| $a_1a_2$ | $b_1b_2$ | $c_1c_2$ | $\cdots$ |
| $a_0a_2$ | $b_0b_2$ | $c_0c_2$ | $\cdots$ |
| $a_1a_2$ | $b_1b_2$ | $c_1c_2$ | $\cdots$ |

(b) Multiple vertical workspaces

Note: the example consider $x_0x_1x_2$ at the points $(a_0, a_1, a_2)$, $(b_0, b_1, b_2)$, $(c_0, c_1, c_2)$, $\ldots$.

Figure 20: The sequence of steps in evaluating one monomial and its derivatives for three paths

| | $x_1x_2x_3x_4$ and its four derivatives evaluated | | |
|---|---|---|---|
| | path 0 | path 1 | path 2 |
| 0 | $a_1$ | $b_1$ | $c_1$ |
| 1 | $a_1 \star a_2$ | $b_1 \star b_2$ | $c_1 \star c_2$ |
| 2 | $a_1a_2 \star a_3$ | $b_1b_2 \star b_3$ | $c_1c_2 \star c_3$ |
| 7 | $a_1a_2a_3 \star a_4$ | $b_1b_2b_3 \star b_4$ | $c_1c_2c_3 \star c_4$ |
| 6 | $a_1a_2 \star a_4$ | $b_1b_2 \star b_4$ | $c_1c_2 \star c_4$ |
| 3 | $a_3 \star a_4$ | $b_3 \star b_4$ | $c_3 \star c_4$ |
| 4 | $a_1 \star a_3a_4$ | $b_1 \star b_3b_4$ | $c_1 \star c_3c_4$ |
| 5 | $a_2 \star a_3a_4$ | $b_2 \star b_3b_4$ | $c_2 \star c_3c_4$ |

Note: three paths have different points $(a_1, a_2, a_3, a_4)$, $(b_1, b_2, b_3, b_4)$, and $(c_1, c_2, c_3, c_4)$. Each new multiplication is marked by a $\star$.

To sum the same element in Jacobian matrix, multiple evaluations follow the same instructions to access the same position of their own workspace. If the monomial workspaces join horizontally like 20(a), we need to sum by column, which causes random memory access. But if we can organize the workspaces vertically like 20(b), the same position of the workspaces are aligned, and we can sum by rows, which creates memory coalescing.

To generate vertical monomial workspaces, we can make all threads of the same block to evaluate one monomial of multiple paths. Reverse mode works directly, because all threads read and write sequentially at the same step. Also, all threads in each block shared the same instructions to evaluate monomials, which save the instruction reading time. An example of multiple evaluations is displayed in Figure 20.

**Algorithm 3** Multiple polynomial evaluations on GPUs

1: **procedure** GPU_PED_MULT($Inst, W$)
2: launch kernel TRANSPOSE_ARRAY($W.\mathbf{x}\_array$, $W.\mathbf{x}\_vertical$, $W.t\_array$, $W.t\_mult$, $W.path\_idx$, $W.\mathbf{x}\_t\_idx$)
3: launch kernel PED_COEF_MULT($Inst.coef$, $W.t\_array$, $W.coef\_mult$)
4: launch kernel PED_MON_MULT($Inst.mon$, $W.coef\_mult$, $W.mon\_mult$)
5: launch kernel PED_SUM_MULT($Inst.sum$, $W.mon\_mult$, $W.matrix\_vertical$)
6: launch kernel TRANSPOSE_ARRAY($W.matrix\_vertical$, $W.matrix$,$W.path\_idx$)
7: **end procedure**

Compared with the tree mode, this consecutive mode has more memory bandwidth. Although monomial evaluation part has twice memory access than tree mode, summation has more speedup due to consecutive memory. See Figure 21. Also, multiple threads in a single block use the same instruction to avoid redundant reading. Thus, this consecutive mode is more suitable for evaluation of multiple paths.

Figure 21: Memory bandwidth of 1,000 evaluations of the same polynomial system(GB/s)

| | name | double | double double | quad double |
|---|---|---|---|---|
| Mon | `cyclic10` | 190.41 | 124.78 | 25.70 |
| | `nash8` | 206.68 | 143.30 | 27.62 |
| | `pieri44` | 209.47 | 147.31 | 27.32 |
| Sum | `cyclic10` | 104.91 | 126.63 | 123.13 |
| | `nash8` | 121.38 | 128.52 | 126.56 |
| | `pieri44` | 87.26 | 80.41 | 77.56 |

Note: details of these polynomial systems are in Section 2.5.2

## 2.5 Computational Results

In this section we report timings and speedups. We implemented the path tracker with the gcc compiler and version 6.5 of the CUDA Toolkit. Our NVIDIA Tesla K20C, which has 2496 cores with a clock speed of 706 MHz, is hosted by a Red Hat Enterprise Linux workstation of Microway, with Intel Xeon E5-2670 processors at 2.6 GHz. Our code is compiled with the optimization flag `-O2`. The settings also are also used in the tests in the other chapters.

### 2.5.1 Test Problems

We selected three examples of polynomial systems, which arose in different applications. The examples can be formulated for any number of equations and variables. Below is a brief description of each system:

#### 2.5.1.1 Cyclic $n$-roots

Our first test problem is the cyclic $n$-roots problem, denoted by $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, $\mathbf{f} = (f_1, f_2, \ldots, f_n)$, with

$$
\begin{aligned}
f_1 &= x_0 + x_1 + \cdots + x_{n-1}, \\
f_2 &= x_0x_1 + x_1x_2 + \cdots + x_{n-2}x_{n-1} + x_{n-1}x_0, \\
f_i &= \sum_{j=0}^{n-1} \prod_{k=j}^{j+i-1} x_{k \bmod n}, i = 3, 4, \ldots, n-1, \\
f_n &= x_0x_1x_2 \cdots x_{n-1} - 1.
\end{aligned}
\tag{2.9}
$$

#### 2.5.1.2 Pieri hypersurface problems

Our second class of test problems has its origin in the output pole placement problem in the control of linear systems. We may view this problem as an inverse eigenvalue problem (34). The polynomial equations arise from minor expansions on

$$\det(A|X) = 0, \quad A \in \mathbb{C}^{n \times m}, \tag{2.10}$$

and where $X$ is an $n$-by-$p$ matrix ($m + p = n$) of unknowns. For example, a 2-plane in complex 4-space (or equivalently, a line in projective 3-space) is represented as

$$X = \begin{bmatrix} 1 & 0 \\ x_{2,1} & 1 \\ x_{2,2} & x_{3,2} \\ 0 & x_{4,2} \end{bmatrix}. \tag{2.11}$$

To determine for the four unknowns in $X$ we need four equations as in (Equation 2.10), which via expansion results in four quadratic equations.

#### 2.5.1.3 Nash equilibrium problems

In game theory, the Nash equilibrium is a solution concept of a competitive game involving two or more players. The solutions of this system give all totally mixed Nash equilibria in a game with $n$ players, where each player has two pure strategies. See (16; 42; 51) for details.

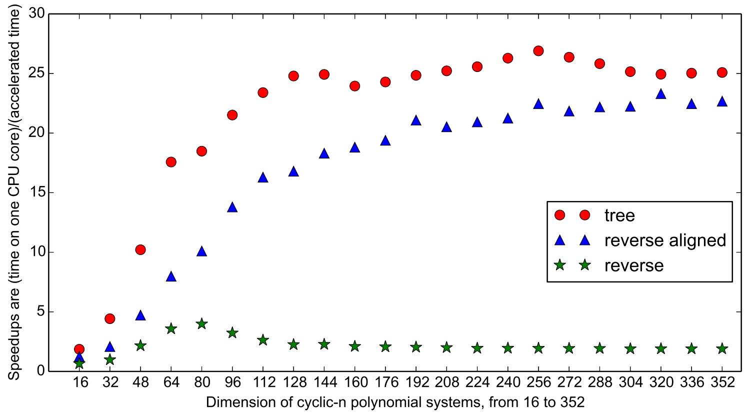
### 2.5.2 Running the single polynomial evaluation on GPUs

Cylic $n$-roots are the testing polynomial systems for reverse mode and tree mode. Cyclic-$n$ polynomial system has $n$ equations, each equation has $n$ monomials and each monomial has 1 to $n$ variables. It is a common benchmark problem in computational algebraic geometry. We evaluate cyclic $n$-roots of dimension 16 to 352 in multi-precision.

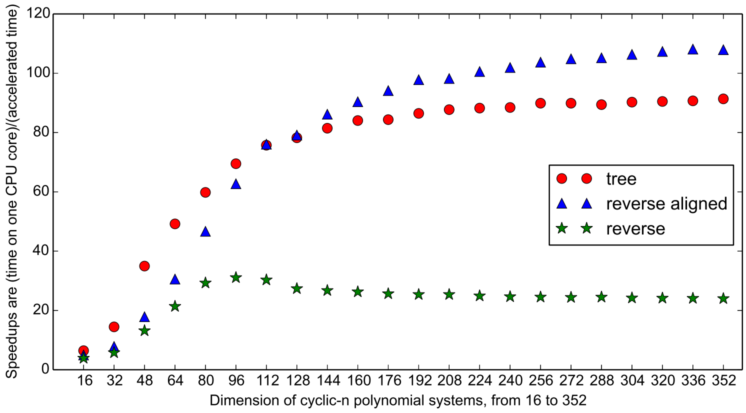
For double precision in Figure 22 (a), the tree mode achieves better speedup. The peak is around dimension 128 and 256, because these dimensions are multiple times of the block size. In the tree mode, the first level has less computation jobs if the number of variables is between $2^n + 1$ to $2^{n+1}$. See Figure 15.

As discussed in Section 2.2.3, tree mode works better for lower precision like complex double, which is memory bounded. For complex double-double and complex quad-double, which are more computation intensive, reverse mode is more efficient. See Figure 22.

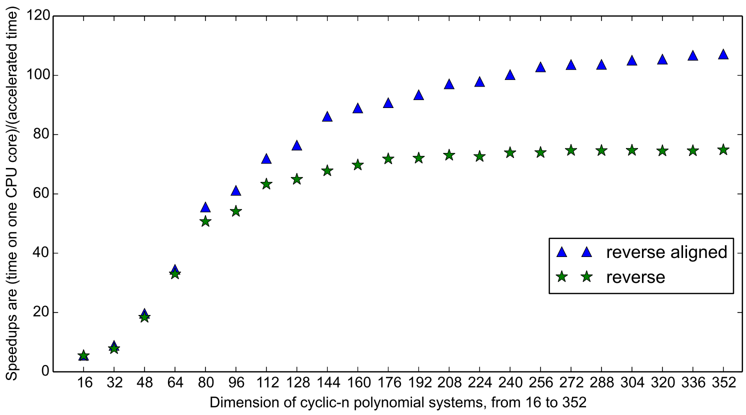
Figure 22: Speedup comparison of tree mode and reverse mode for cyclic $n$-roots




(a) Complex double precision




(b) Complex double double precision



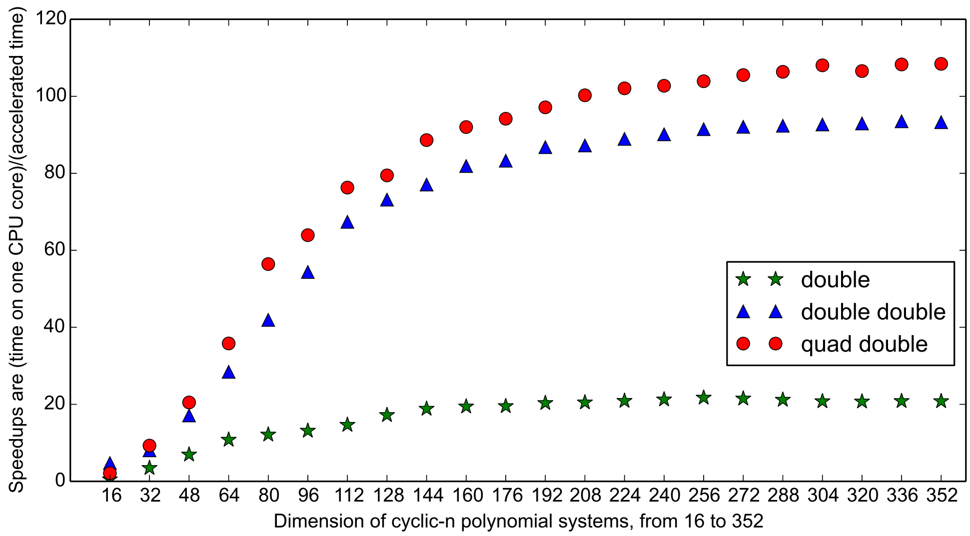

(c) Complex quad double precision

TABLE I: Speedup for one evaluation and differentiation of cyclic $n$-roots in various precisions and in various dimensions.

| | complex double | | | complex double double | | | complex quad double | | |
|---|---|---|---|---|---|---|---|---|---|
| $n$ | cpu | gpu | S | cpu | gpu | S | cpu | gpu | S |
| 16 | 0.028 | 0.055 | 0.509 | 0.693 | 0.145 | 4.767 | 2.374 | 1.123 | 2.113 |
| 32 | 0.267 | 0.077 | 3.464 | 2.148 | 0.267 | 8.054 | 19.553 | 2.106 | 9.285 |
| 48 | 0.814 | 0.117 | 6.948 | 7.160 | 0.418 | 17.118 | 66.936 | 3.267 | 20.490 |
| 64 | 1.898 | 0.176 | 10.796 | 16.900 | 0.594 | 28.447 | 159.186 | 4.445 | 35.810 |
| 80 | 3.348 | 0.276 | 12.135 | 32.902 | 0.785 | 41.929 | 313.973 | 5.564 | 56.425 |
| 96 | 5.395 | 0.411 | 13.142 | 57.791 | 1.063 | 54.373 | 558.524 | 8.737 | 63.927 |
| 112 | 8.676 | 0.592 | 14.666 | 97.866 | 1.452 | 67.393 | 897.442 | 11.764 | 76.285 |
| 128 | 14.016 | 0.815 | 17.204 | 148.108 | 2.025 | 73.158 | 1340.249 | 16.869 | 79.451 |
| 144 | 21.681 | 1.151 | 18.842 | 209.462 | 2.718 | 77.072 | 1912.814 | 21.586 | 88.613 |
| 160 | 30.123 | 1.551 | 19.426 | 287.067 | 3.506 | 81.882 | 2630.132 | 28.587 | 92.003 |
| 176 | 39.663 | 2.032 | 19.520 | 383.578 | 4.607 | 83.266 | 3497.826 | 37.148 | 94.160 |
| 192 | 52.669 | 2.592 | 20.322 | 499.784 | 5.759 | 86.788 | 4545.077 | 46.797 | 97.123 |
| 208 | 66.725 | 3.257 | 20.484 | 637.110 | 7.304 | 87.232 | 5772.866 | 57.580 | 100.257 |
| 224 | 83.004 | 3.974 | 20.889 | 797.452 | 8.967 | 88.932 | 7206.397 | 70.608 | 102.062 |
| 240 | 102.646 | 4.828 | 21.261 | 980.851 | 10.884 | 90.122 | 8852.913 | 86.181 | 102.725 |
| 256 | 124.910 | 5.750 | 21.725 | 1191.949 | 13.033 | 91.460 | 10722.170 | 103.191 | 103.906 |
| 272 | 149.886 | 6.974 | 21.492 | 1431.843 | 15.552 | 92.070 | 12875.790 | 122.041 | 105.504 |
| 288 | 176.960 | 8.355 | 21.179 | 1700.141 | 18.411 | 92.342 | 15230.040 | 143.201 | 106.354 |
| 304 | 207.467 | 9.980 | 20.788 | 2000.403 | 21.584 | 92.681 | 17898.370 | 165.669 | 108.037 |
| 320 | 242.306 | 11.693 | 20.721 | 2326.944 | 25.036 | 92.943 | 20864.140 | 195.818 | 106.548 |
| 336 | 280.838 | 13.476 | 20.839 | 2693.531 | 28.808 | 93.500 | 24106.570 | 222.668 | 108.262 |
| 352 | 322.295 | 15.489 | 20.808 | 3091.023 | 33.141 | 93.268 | 27692.870 | 255.431 | 108.416 |

Note: The last column for each dimension and precision contains the speedup S. Timing in milliseconds.

Figure 23: Speedup for one evaluation and differentiation of cyclic $n$-roots in various precisions




### 2.5.3 Running multiple polynomial evaluations on GPUs

For testing multiple monomial evaluation, we choose polynomial systems of relative small dimension, in order to prove that we can achieve good speedups as long as we compute many PEDs for the same polynomial systems.

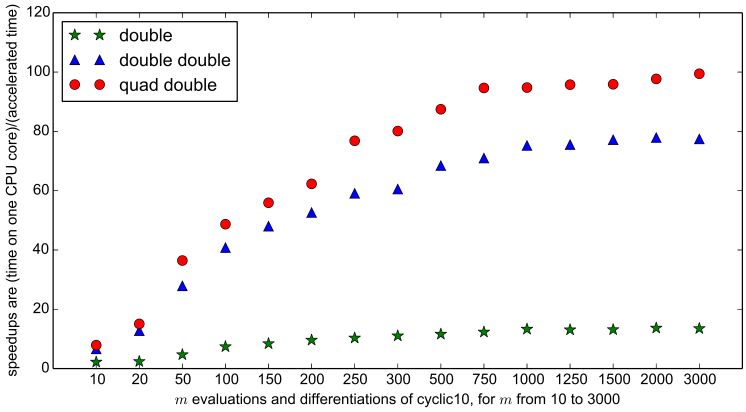
1. `cyclic10:` the cyclic 10-roots problem is a 10-dimensional system with 34,940 isolated complex solutions. Except for the last equation (which has two terms), every polynomial has 10 monomials. The $k$-th polynomial in this system is of degree $k$. These roots appear in the study of complex Hadamard matrices (52).

2. `pieri44`: there are 24,024 four dimensional planes that meet 16 four dimensional planes, given in general position. This system is a 16-dimensional problem and can be interpreted as a matrix completion problem (34), see also (30; 31). Every polynomial in the system is of degree 4 and has 246 monomials.

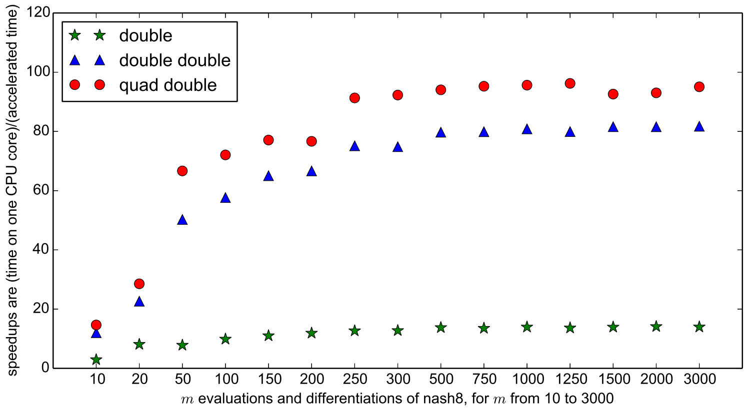
3. `nash8`: the solutions of this system give all totally mixed Nash equilibria in a game with 8 players. For generic payoff matrices, this 8-dimensional system has 14,833 equilibria. Every polynomial in this system has 130 monomials of degrees ranging from one till seven.

Table II, Table III and Table IV show the running times of CPU and GPU in multiple precision. $m$ is the number of Newton iterations to get convergent solutions. Figure 24 visualize the speedups in these tables.

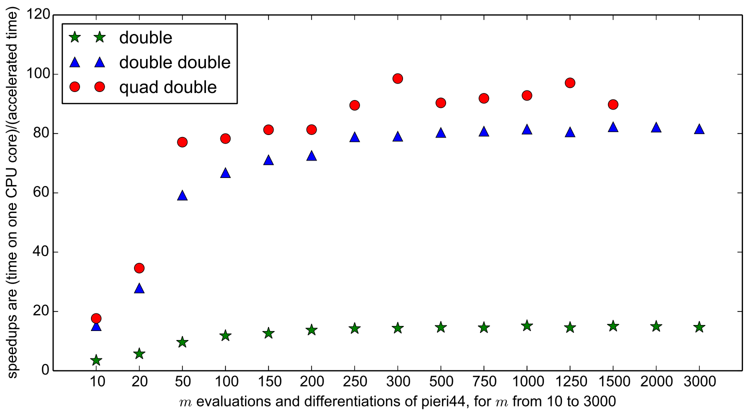
Figure 24: Speedups for multiple evaluations and differentiations of three polynomial systems




(a) cyclic 10-roots




(b) Nash equilibrium system




(c) hypersurface Pieri system

Note: All of them are computed in complex double, double double, and quad double arithmetic.

TABLE II: Speedups for multiple evaluations and differentiations of the cyclic 10-roots problem.

complex double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 0.062 | 0.017 | 0.008 | 0.004 | 0.028 | 2.19 |
| 20 | 0.078 | 0.020 | 0.008 | 0.004 | 0.033 | 2.39 |
| 50 | 0.188 | 0.024 | 0.011 | 0.005 | 0.040 | 4.69 |
| 100 | 0.379 | 0.030 | 0.016 | 0.006 | 0.051 | 7.39 |
| 200 | 0.732 | 0.042 | 0.026 | 0.008 | 0.076 | 9.60 |
| 500 | 1.824 | 0.087 | 0.056 | 0.015 | 0.157 | 11.61 |
| 1000 | 3.748 | 0.155 | 0.101 | 0.026 | 0.282 | 13.30 |
| 2000 | 7.381 | 0.299 | 0.191 | 0.050 | 0.540 | 13.67 |
| 3000 | 11.148 | 0.459 | 0.284 | 0.082 | 0.826 | 13.50 |

complex double double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 0.587 | 0.066 | 0.011 | 0.011 | 0.088 | 6.65 |
| 20 | 1.135 | 0.066 | 0.012 | 0.011 | 0.089 | 12.79 |
| 50 | 2.808 | 0.072 | 0.017 | 0.012 | 0.101 | 27.90 |
| 100 | 5.598 | 0.092 | 0.028 | 0.017 | 0.137 | 40.81 |
| 200 | 11.225 | 0.145 | 0.043 | 0.025 | 0.213 | 52.64 |
| 500 | 27.912 | 0.263 | 0.092 | 0.052 | 0.408 | 68.47 |
| 1000 | 55.871 | 0.472 | 0.175 | 0.096 | 0.743 | 75.24 |
| 2000 | 112.040 | 0.917 | 0.338 | 0.183 | 1.438 | 77.92 |
| 3000 | 167.568 | 1.383 | 0.502 | 0.278 | 2.163 | 77.47 |

complex quad double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 5.572 | 0.632 | 0.042 | 0.072 | 0.705 | 7.91 |
| 20 | 11.129 | 0.622 | 0.043 | 0.073 | 0.738 | 15.07 |
| 50 | 27.769 | 0.633 | 0.054 | 0.075 | 0.762 | 36.44 |
| 100 | 55.566 | 0.931 | 0.080 | 0.130 | 1.141 | 48.70 |
| 200 | 111.027 | 1.438 | 0.120 | 0.224 | 1.782 | 62.29 |
| 500 | 277.978 | 2.486 | 0.257 | 0.436 | 3.178 | 87.46 |
| 1000 | 554.742 | 4.582 | 0.485 | 0.786 | 5.853 | 94.77 |
| 2000 | 1111.412 | 8.916 | 0.929 | 1.532 | 11.377 | 97.69 |
| 3000 | 1676.977 | 13.244 | 1.375 | 2.245 | 16.864 | 99.44 |

Note: timing in milliseconds

TABLE III: Speedups for multiple evaluations and differentiations of the Nash equilibrium system

complex double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 0.311 | 0.042 | 0.050 | 0.015 | 0.106 | 2.92 |
| 20 | 0.586 | 0.057 | 0.069 | 0.015 | 0.072 | 8.10 |
| 50 | 1.417 | 0.079 | 0.075 | 0.027 | 0.181 | 7.81 |
| 100 | 2.813 | 0.140 | 0.113 | 0.032 | 0.285 | 9.86 |
| 200 | 5.586 | 0.244 | 0.169 | 0.057 | 0.470 | 11.89 |
| 500 | 13.834 | 0.567 | 0.314 | 0.125 | 1.006 | 13.75 |
| 1000 | 27.509 | 1.111 | 0.608 | 0.254 | 1.973 | 13.94 |
| 2000 | 55.157 | 2.209 | 1.179 | 0.523 | 3.910 | 14.11 |
| 3000 | 82.710 | 3.303 | 1.742 | 0.877 | 5.922 | 13.97 |

complex double double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 4.345 | 0.195 | 0.116 | 0.050 | 0.361 | 12.03 |
| 20 | 8.664 | 0.201 | 0.125 | 0.056 | 0.382 | 22.66 |
| 50 | 21.587 | 0.226 | 0.141 | 0.062 | 0.429 | 50.26 |
| 100 | 43.239 | 0.411 | 0.219 | 0.120 | 0.750 | 57.68 |
| 200 | 86.489 | 0.762 | 0.321 | 0.215 | 1.297 | 66.67 |
| 500 | 216.220 | 1.623 | 0.598 | 0.491 | 2.712 | 79.74 |
| 1000 | 431.826 | 3.203 | 1.182 | 0.957 | 5.341 | 80.86 |
| 2000 | 864.464 | 6.361 | 2.299 | 1.936 | 10.596 | 81.58 |
| 3000 | 1301.577 | 9.517 | 3.420 | 2.984 | 15.922 | 81.75 |

complex quad double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 43.425 | 1.956 | 0.502 | 0.506 | 2.964 | 14.65 |
| 20 | 86.566 | 1.977 | 0.522 | 0.534 | 3.033 | 28.55 |
| 50 | 216.214 | 2.154 | 0.552 | 0.537 | 3.244 | 66.66 |
| 100 | 433.039 | 4.150 | 0.807 | 1.051 | 6.009 | 72.07 |
| 200 | 866.149 | 8.051 | 1.171 | 2.077 | 11.299 | 76.66 |
| 500 | 2161.734 | 16.866 | 1.938 | 4.182 | 22.986 | 94.05 |
| 1000 | 4327.603 | 33.228 | 3.852 | 8.173 | 45.253 | 95.63 |
| 2000 | 8652.404 | 68.903 | 7.380 | 16.727 | 93.010 | 93.03 |
| 3000 | 12977.386 | 100.940 | 10.799 | 24.771 | 136.510 | 95.07 |

Note: timing in milliseconds

TABLE IV: Speedups for multiple evaluations and differentiations of the Pieri hypersurface system

complex double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 1.129 | 0.137 | 0.138 | 0.049 | 0.324 | 3.48 |
| 20 | 2.127 | 0.168 | 0.156 | 0.050 | 0.373 | 5.70 |
| 50 | 5.223 | 0.239 | 0.208 | 0.097 | 0.544 | 9.60 |
| 100 | 10.226 | 0.447 | 0.306 | 0.113 | 0.866 | 11.80 |
| 200 | 20.239 | 0.794 | 0.475 | 0.206 | 1.475 | 13.72 |
| 500 | 50.778 | 1.890 | 1.113 | 0.471 | 3.474 | 14.62 |
| 750 | 75.665 | 2.895 | 1.589 | 0.729 | 5.213 | 14.51 |
| 1000 | 102.170 | 3.718 | 2.074 | 0.958 | 6.751 | 15.13 |
| 2000 | 201.537 | 7.425 | 4.064 | 2.003 | 13.492 | 14.94 |
| 3000 | 302.158 | 11.108 | 6.138 | 3.351 | 20.597 | 14.67 |

complex double double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 15.116 | 0.559 | 0.266 | 0.170 | 0.995 | 15.19 |
| 20 | 29.886 | 0.582 | 0.287 | 0.202 | 1.071 | 27.91 |
| 50 | 75.020 | 0.659 | 0.391 | 0.217 | 1.267 | 59.22 |
| 100 | 151.854 | 1.263 | 0.573 | 0.437 | 2.273 | 66.80 |
| 200 | 298.554 | 2.425 | 0.907 | 0.781 | 4.113 | 72.59 |
| 500 | 746.392 | 5.299 | 2.129 | 1.862 | 9.289 | 80.35 |
| 1000 | 1491.030 | 10.570 | 4.080 | 3.649 | 18.299 | 81.48 |
| 2000 | 2990.387 | 21.057 | 7.908 | 7.429 | 36.394 | 82.17 |
| 3000 | 4478.135 | 31.423 | 12.001 | 11.455 | 54.879 | 81.60 |

complex quad double arithmetic

| | CPU | GPU | | | | |
|---|---|---|---|---|---|---|
| #evals | total | mon | sum | coeff | total | speedup |
| 10 | 146.920 | 5.329 | 1.132 | 1.867 | 8.328 | 17.64 |
| 20 | 293.975 | 5.369 | 1.188 | 1.935 | 8.493 | 34.61 |
| 50 | 734.441 | 6.104 | 1.954 | 1.468 | 9.526 | 77.10 |
| 100 | 1470.332 | 12.760 | 2.123 | 3.895 | 18.778 | 78.30 |
| 200 | 2942.909 | 25.181 | 3.149 | 7.859 | 36.189 | 81.32 |
| 500 | 7346.943 | 58.511 | 6.909 | 15.901 | 81.321 | 90.35 |
| 1000 | 14697.217 | 113.970 | 13.027 | 31.309 | 158.306 | 92.84 |
| 1250 | 18394.761 | 134.100 | 16.487 | 38.865 | 189.452 | 97.09 |
| 1500 | 22045.021 | 177.920 | 19.023 | 48.569 | 245.512 | 89.79 |

Note: timing in milliseconds

### 2.5.4 Conclusion

In this chapter, parallel algorithms are given to polynomial evaluation and differentiation (PED) on GPUs. The problem is split into three parts: the evaluation of homotopy coefficients, the evaluation and differentiation of monomials, and the summation to the Jacobian matrix.

For the single PED, the tree mode works better for lower precision like complex double, which is memory bounded. For complex double-double and complex quad-double, which are more computation intensive, reverse mode with aligned memory is more efficient.

Multiple PEDs can be computed simultaneously following the same instruction. The data structure is reorganized vertically for all evaluations, so there is more memory coalescing in monomial and summation kernels. With many PEDs, even for a small system like cylic-10, the speedup is better than that of a single large dimension polynomial system.

# CHAPTER 3

# NEWTON'S METHOD ON GPUS

In this chapter, we design accelerated algorithms for solving large polynomial systems with numerical methods, Newton's method. The ideas are originally presented in our paper (58).

## 3.1 Overview

Newton's method is a numerical method to solve nonlinear systems. Given a polynomial system $\mathbf{f}(\mathbf{x})$, we begin with a start point $\mathbf{x}_0$ and find better approximations successively by:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{f}'(\mathbf{x}_k)]^{-1}\mathbf{f}(\mathbf{x}_k), \text{ for } k = 0, 1, \ldots \tag{3.1}$$

If $\mathbf{x}_0$ is close to a solution $\alpha$ and $\mathbf{f}'(x) \neq 0$ near the solution, then the rate of convergence is quadratic.

In Newton's method, each iteration are two major steps:

1. evaluate and differentiate $\mathbf{f}(\mathbf{x})$,
2. solve the linear equations $\mathbf{f}'(\mathbf{x}_k)\Delta\mathbf{x}_k = \mathbf{f}(\mathbf{x}_k)$

then we update $\mathbf{x}$ by $\mathbf{x}_{k+1} = \mathbf{x}_k - \Delta\mathbf{x}_k$. Repeat iterations until a sufficient accurate solution is reached.

For the linear solver, we choose Modified Gram-Schmidt(MGS), because it can solve overdetermined matrices like our applications in Section 4.3.

## 3.2 Check the convergence of Newton iteration

Newton iterations are not always convergent, and the reasons of failures include arithmetic precision limits more accurate approximation, $\mathbf{x}_k$ is not in the convergent range, etc. To check the status of convergence, residues $\| \mathbf{f}(\mathbf{x}_k) \|$ and correction size $\| \Delta x_k \|$ can be used for each Newton iteration. Some checking standards are listed in Figure 25.

Figure 25: Standards to check the convergence of Newton iterations

1. $\| \mathbf{f}(\mathbf{x}_k) \|$ and its ratio to $\| x_k \|$. If either of them is small enough, $x_k$ is sufficient accurate.
2. $\| \Delta\mathbf{x}_k \|$ and its ratio to $\| \mathbf{x}_k \|$. If either of them is small enough, $\mathbf{x}_k$ is sufficient accurate.
3. If $\| \mathbf{f}(\mathbf{x}_k) \| > \| \mathbf{f}(\mathbf{x}_{k-1}) \|$, the updated approximation $\mathbf{x}_k$ is divergent.
4. If $\| \Delta x_k \| > \| \Delta x_{k-1} \|$, it implies that Newton's method is not convergent quadratically. This could be used for more restrict convergence.

## 3.3 Design Newton's method on GPUs

Newton's method is sequential, and each iteration depends on the result of the last one. After each iteration, the CPU host requests the control parameters from the GPU device. These control parameters include $\| \mathbf{f}(\mathbf{x}_k) \|$, $\| \Delta x_k \|$ and $\| x_k \|$. These control parameters are not necessary to be extreme accurate, so double part is sufficient to represent for double double and quad double. With communications of these 3 double variables between host and device, the host can control the device kernel launches and finish Newton's method. The host only costs $O(n\_iteration)$ time much less than the device. We design GPU algorithms for Newton's method in different versions.

### 3.3.1 Newton's method to find one solution

To find one solution of the polynomial system, we can pick a random point, typically form a unit circle as the start point. Without any constrain, we allow points to walk through space and fall into the convergent range of any solution. Thus, we can run many Newton iterations, until standard 1 or 2 of Figure 25 is satisfied. See Algorithm 4.

**Algorithm 4** Newton's method for find one solution

```
procedure GPU_Newton(Inst, W, P)
    last_max_eq_val ← P.max_eq_val
    for k = 1 to P.max_iteration do
        GPU_PED(Inst,W)
        launch kernel Max_Array(W.eq_val, max_eq_val)
        copy max_eq_val from device to host
        launch kernel Max_Array(W.x, max_x)
        copy max_x from device to host
        if max_eq_val < P.tolerance or max_eq_val/max_x < P.tolerance then
            return success
        end if
        GPU_MGS(W)
        launch kernel Update_x(W.x,W.Δx)
        launch kernel Max_Array(W.Δx, max_Δx)
        copy max_Δx from device to host
        if max_Δx < P.tolerance or max_Δx/max_x < P.tolerance then
            return success
        end if
        last_max_eq_val ← max_eq_val
    end for
    return fail
end procedure
```

### 3.3.2 Newton's method for path tracking

To track a solution path, we avoid jumping from one solution path to others, so we limit number of Newton's iteration (3 for double) and use 3 to check each iteration, until 1 or 2 is satisfied. See Algorithm 5.

**Algorithm 5** An accelerated Newton's method in path tracking

```
procedure GPU_NEWTON_PATH(Inst, W, P)
    last_max_eq_val ← P.max_eq_val
    for k = 1 to P.max_iteration do
        GPU_PED(Inst,W)
        launch kernel MAX_ARRAY(W.eq_val, max_eq_val)
        copy max_eq_val from device to host
        if max_eq_val > last_max_eq_val then
            return fail
        end if
        launch kernel MAX_ARRAY(W.x, max_x)
        copy max_x from device to host
        if max_eq_val < P.tolerance or max_eq_val/max_x < P.tolerance then
            return success
        end if
        GPU_MGS(W)
        launch kernel UPDATE_x(W.x,W.Δx)
        launch kernel MAX_ARRAY(W.Δx, max_Δx)
        copy max_Δx from device to host
        if max_Δx < P.tolerance or max_Δx/max_x < P.tolerance then
            return success
        end if
        last_max_eq_val ← max_eq_val
    end for
    return fail
end procedure
```

### 3.3.3 Newton's method for refining the solution

To refine a solution, we want the result to be as accurate as possible. To increase its accuracy, we run more iterations (5 for double), until 3 is not satisfied any more. Then we choose the point before the last correction $\mathbf{x}_{k-1}$ , which has minimal residue $\| \mathbf{f}(\mathbf{x}_{k-1}) \|$. See Algorithm 6.

**Algorithm 6** An accelerated Newton's method for refinement

```
procedure GPU_NEWTON_REFINE(Inst, W, P)
    GPU_PED(Inst,W)
    launch kernel MAX_ARRAY(W.eq_val, max_eq_val)
    copy max_eq_val from device to host
    last_max_eq_val ← P.max_eq_val
    for k = 1 to P.max_iteration do
        GPU_MGS(W)
        swap the pointers of W.last_x and W.x
        launch kernel UPDATE_NEW_x(W.x, W.last_x,W.Δx)
        last_max_eq_val ← max_eq_val
        GPU_PED(Inst,W)
        launch kernel MAX_ARRAY(W.eq_val, max_eq_val)
        copy max_eq_val from device to host
        if max_eq_val > last_max_eq_val then
            swap the pointers of W.last_x and W.x
            max_eq_val = last_max_eq_val
            break
        end if
    end for
    launch kernel MAX_ARRAY(W.Δx, max_Δx)
    copy max_Δx from device to host
    return max_eq_val, max_Δx
end procedure
```

## 3.4 Computational Results

In this section we report timings and speedups.

### 3.4.1 The Chandrasekhar H-Equation

The system arises from the discretization of an integral equation. The problem was treated with Newton's method in (33). In (21), the system was studied with methods in computer algebra. We follow the formulation in (21):

$$\begin{aligned} & f_i(H_1, H_2, \ldots, H_n) \\ & \quad = 2nH_i - cH_i \left( \sum_{j=0}^{n-1} \frac{i}{i+j} H_j \right) - 2n = 0, \end{aligned} \tag{3.2}$$

for $i = 1, 2, \ldots, n$, and for some constant $c$, $0 < c \leq 1$. As the evaluation and differentiation cost is linear in $n$, the cost of Newton's method is dominated by the cost for solving the linear system, which is $O(n^3)$.

For all $c$, there is one real solution with all its components positive and relatively close to 1. Starting at $H_i = 1$ for all $i$ leads to a quadratically convergent Newton's method. The value for the parameter $c$ we used in our experiments is 33/64.

Table V shows the running times obtained with the command `time`. Comparing absolute real wall clock times: when we double the dimensions from 2048 to 4096, the accelerated versions of the code run twice as fast, 20 minutes versus 42 minutes without acceleration. As the cost of evaluation and differentiation grows only linearly in $n$, the cost of the linear solving dominates and as the dimension grows, the difference in speedups between the two accelerated versions fades out.

TABLE V: Running six iterations of Newton's method in complex double double arithmetic

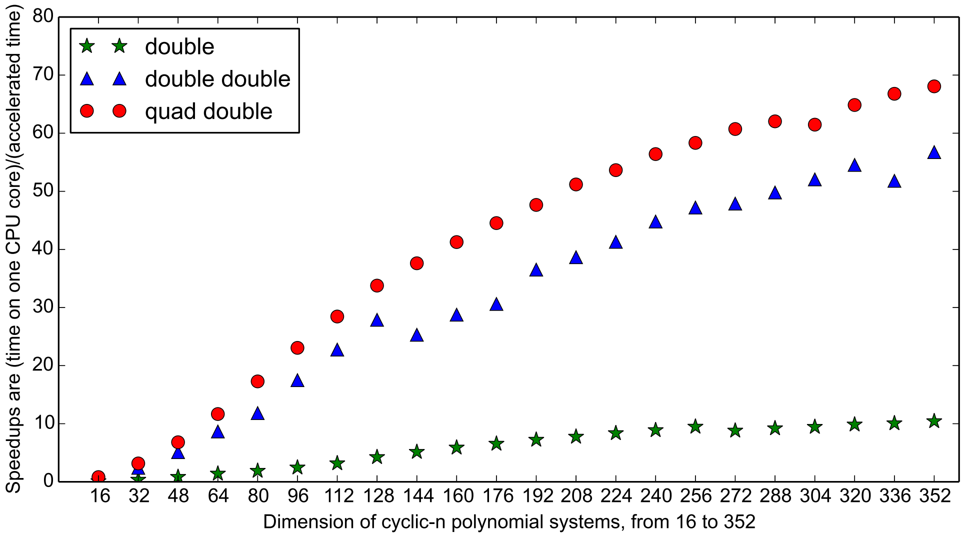
| $n$ | mode | real | user | sys | speedup |
|---|---|---|---|---|---|
| 1024 | CPU | 5m22.360s | 5m21.680s | 0.139s | |
| | GPU1 | 24.074s | 18.667s | 5.203s | 13.39 |
| | GPU2 | 20.083s | 11.564s | 8.268s | 16.05 |
| 2048 | CPU | 42m41.597s | 42m37.236s | 0.302s | |
| | GPU1 | 2m45.084s | 1m48.502s | 56.175s | 15.52 |
| | GPU2 | 2m29.770s | 1m26.373s | 1m03.014s | 17.10 |
| 3072 | CPU | 144m13.978s | 144m00.880s | 0.216s | |
| | GPU1 | 8m50.933s | 5m34.427s | 3m15.608s | 16.30 |
| | GPU2 | 8m15.565s | 4m43.333s | 3m31.362s | 17.46 |
| 4096 | CPU | 340m00.724s | 339m27.019s | 0.929s | |
| | GPU1 | 20m26.989s | 13m39.416s | 6m45.799s | 16.63 |
| | GPU2 | 19m24.243s | 11m01.558s | 8m20.698s | 17.52 |

Note: it runs by one core on the CPU and accelerated by the K20C with block size equal to 128, once with the evaluation and differentiation done by the CPU (GPU1) and once with all computations on the GPU (GPU2).

### 3.4.2 Running one Newton's method of cyclic $n$-roots

Cylic $n$-roots in Section 2.5.1.1 are another testing polynomial systems. The dimension $n$ goes from 16 to 352. Table VI shows the running times of CPU and GPU in multiple precision. $m$ is the number of Newton iterations to get convergent solutions. Figure 26 visualize the speedups in Table VI. As the cost of PED and MGS both grow in $O(n^3)$, we have similar speedups as that of PED in Section 2.5.2 .

Figure 26: Running one Newton's method for cyclic $n$-roots in various precisions and in various dimensions.




### 3.4.3 Conclusion

In this chapter, we design accelerated algorithms for Newton's method. For each Newton iteration, we combined two computational intensive steps, PED and MGS, on GPUs. After each iteration, the CPU host control the GPU device by the 3 double control parameters from the device. Both speed up and quality up are achieved with acceleration on GPUs.

TABLE VI: Running one Newton's method for cyclic $n$-roots in various precisions and in various dimensions.

| | complex double | | | | complex double double | | | | complex quad double | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $m$ | cpu | gpu | S | $m$ | cpu | gpu | S | $m$ | cpu | gpu | S |
| 16 | 3 | 0.15 | 1.55 | 0.10 | 5 | 2.82 | 4.72 | 0.60 | 6 | 31.24 | 38.38 | 0.81 |
| 32 | 3 | 0.92 | 2.61 | 0.35 | 4 | 17.35 | 7.21 | 2.41 | 5 | 204.99 | 64.75 | 3.17 |
| 48 | 3 | 3.25 | 3.88 | 0.84 | 5 | 70.76 | 13.86 | 5.11 | 7 | 935.22 | 137.30 | 6.81 |
| 64 | 3 | 7.32 | 5.22 | 1.40 | 5 | 166.78 | 19.25 | 8.66 | 6 | 1909.63 | 163.68 | 11.67 |
| 80 | 3 | 13.26 | 7.03 | 1.88 | 4 | 235.26 | 19.89 | 11.83 | 5 | 3161.74 | 182.92 | 17.28 |
| 96 | 3 | 22.08 | 9.00 | 2.45 | 4 | 466.89 | 26.69 | 17.50 | 5 | 5553.42 | 240.86 | 23.06 |
| 112 | 3 | 35.57 | 11.19 | 3.18 | 5 | 935.46 | 41.10 | 22.76 | 6 | 10448.05 | 367.23 | 28.45 |
| 128 | 3 | 58.08 | 13.67 | 4.25 | 4 | 1155.56 | 41.44 | 27.88 | 5 | 13220.62 | 391.49 | 33.77 |
| 144 | 3 | 89.05 | 17.32 | 5.14 | 4 | 1643.32 | 64.93 | 25.31 | 5 | 18825.85 | 500.48 | 37.62 |
| 160 | 3 | 123.51 | 20.98 | 5.89 | 5 | 2735.92 | 95.10 | 28.77 | 6 | 30476.23 | 738.58 | 41.26 |
| 176 | 3 | 165.06 | 25.21 | 6.55 | 5 | 3270.80 | 106.84 | 30.61 | 6 | 40562.41 | 910.85 | 44.53 |
| 192 | 3 | 214.94 | 29.72 | 7.23 | 5 | 4738.31 | 129.74 | 36.52 | 7 | 60654.46 | 1272.72 | 47.66 |
| 208 | 3 | 273.44 | 35.25 | 7.76 | 4 | 4960.92 | 128.33 | 38.66 | 5 | 56610.35 | 1106.07 | 51.18 |
| 224 | 3 | 342.81 | 40.97 | 8.37 | 5 | 7562.59 | 183.10 | 41.30 | 6 | 83375.63 | 1554.22 | 53.64 |
| 240 | 3 | 421.82 | 47.47 | 8.89 | 4 | 7640.55 | 170.55 | 44.80 | 5 | 86844.43 | 1539.55 | 56.41 |
| 256 | 3 | 514.06 | 54.23 | 9.48 | 5 | 11265.36 | 238.63 | 47.21 | 6 | 124071.40 | 2126.90 | 58.33 |
| 272 | 3 | 611.70 | 69.47 | 8.81 | 4 | 11064.50 | 231.06 | 47.89 | 5 | 126159.60 | 2077.61 | 60.72 |
| 288 | 3 | 725.22 | 78.79 | 9.20 | 5 | 16070.59 | 322.71 | 49.80 | 6 | 176611.80 | 2846.56 | 62.04 |
| 304 | 3 | 851.21 | 90.06 | 9.45 | 5 | 18808.85 | 361.37 | 52.05 | 7 | 220810.90 | 3591.97 | 61.47 |
| 320 | 3 | 993.82 | 100.86 | 9.85 | 5 | 21971.36 | 402.82 | 54.54 | 7 | 278637.30 | 4296.27 | 64.86 |
| 336 | 3 | 1148.51 | 114.11 | 10.07 | 5 | 22784.08 | 439.82 | 51.80 | 6 | 279560.70 | 4185.93 | 66.79 |
| 352 | 3 | 1319.96 | 126.62 | 10.42 | 4 | 24002.53 | 422.99 | 56.74 | 5 | 272289.00 | 4000.33 | 68.07 |

Note: The number of Newton iterations equals $m$. The last column for each dimension and precision contains the speedup S. Timing in milliseconds.

# CHAPTER 4

## SINGLE PATH TRACKING ON GPUS

Path tracking is a numerical compute-intensive method. Tracking one single path is sequential. It might take hundreds of steps of prediction and correction. In this section, we first develop the predictor on GPU. Then, we join the predictor and the corrector in Section 3.3.2 as a single path tracker on GPU. The ideas in this chapter are presented in our paper (59).

### 4.1 Predictor

The predictor uses previous points to generate an estimated point that is close enough to the solution path. Because the solution path $\mathbf{x}(t)$ is continuous, we can predict by interpolation. For the solution path of any variable $x(t)$, we use previous $p$ points $\{x(t_0), x(t_1), \ldots, x(t_{p-1})\}$ to predict the new point $\tilde{x}(t)$. By the Newton polynomial,

$$\begin{aligned}\tilde{x}(t) =& x(t_0) + x(t_0, t_1)(t - t_0) + x(t_0, t_1, t_2)(t - t_0)(t - t_1) \\ & + \cdots + x(t_0, t_1, \ldots, t_{p-1})(t - t_0)(t - t_1) \cdots (t - t_{p-2})\end{aligned}$$

where $x(t_i, t_1, \ldots, t_j)$ are divided differences, computed recursively by

$$x(t_i, t_{i+1}, \ldots, t_j) = \frac{x(t_i, t_{i+1}, \ldots, t_j) - x(t_{i+1}, \ldots, t_j)}{t_i - t_j}$$

To compute all divided differences efficiently, we use the following table:

$$
\begin{array}{c|ccccc}
t_0 & x(t_0) & & & & \\
t_1 & x(t_1) & x(t_0,t_1) & & & \\
t_2 & x(t_2) & x(t_1,t_2) & x(t_0,t_1,t_2) & & \\
t_3 & x(t_3) & x(t_2,t_3) & x(t_1,t_2,t_3) & x(t_0,t_1,t_2,t_3) & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

The diagonal is the divided differences we need and intermediate results can be overwritten. Thus, we compute column by column from left to the right, and each column is computed from bottom to the top. During this process, each new element is computed by its left and its upper left elements, and after computation, its left element is not used any more. Thus, the new element can overwrite its left element. So the total space is $p$ to compute the divided differences for each variable.

Then, the new point value can be computed by Horner's method:

$$\tilde{x}(t) = x(t_0) + (t-t_0)(x(t_0,t_1) + (t-t_1)(x(t_0,t_1,t_2) + \cdots + (t-t_{p-2})x(t_0,t_1,\ldots,t_{p-1}))\cdots)$$

For GPU implementation, each thread can handle one variable following the same instruction. Within a block, all threads share the $t$s' values of $\{t_0, t_1, \ldots, t_{p-1}\}$, and they can be preloaded into shared memory. Also, he differences of $t$, $\{t-t_0, t-t_1, \ldots, t-t_{p-2}\}$ and $\{t_i - t_j\}_{0\leq i<j\leq p-1}$, can be precomputed in shared memory, too. Although the size of the second part is $p(p+1)/2$, in our real applications, $p$ is from 2 to 5 and the second part does not take too much shared memory.

Previous points on a solution path and $t$s are stored in global memory. Suppose $p$ is numbers of previous points used by the predictor, the total space is $(p+1)dim$. For each step, the new point overwrite the first one of the array of the previous points to reuse the space. In this case, an alternative pointer is used to identify the newest point.

## 4.2 Single path tracking on GPUs

Single path tracking is a sequential algorithm to follow the solution path from the start solution to the target solution. For each step in path tracking, it consists of a predictor, a corrector and a step controller.

1. Predictor: Each variable has an independent interpolation. $O(dim * n\_predict^2)$
2. Corrector: Newton's method, each Newton's iteration consists of the following:
   (a) Polynomial evaluation and differentiation. Depends on systems, cyclic-n: $O(dim^3)$
   (b) Linear solver by Modified Gram-Schmidt. $O(dim^3)$
   (c) Convergence check
      i. Evaluate $\| \mathbf{f}(\mathbf{x}_k) \|$, $\| \Delta x_k \|$ and $\| x_k \|$. $O(dim)$
      ii. Compare with error tolerance or last step. $O(1)$
3. Step controller: If correct is fail, decrease $\Delta t$. If success, increase $t$ by $\Delta t$, until $t = 1$. If success several steps, increase $\Delta t$ before adding to $t$. O(1)

From complexity analysis, predictor (1), polynomial evaluation and differentiation (2.(a)) and Modified Gram-Schmidt (2.(b)) are compute-intensive parts can be run on GPUs device.

Also, the evaluation part of convergence check (2.(c).i) can be done on GPUs device. For $O(1)$ part, the step controller and comparison part of convergence check (2.(c).ii). In the step controller, $t$ and $\Delta t$ are controlled by the host, and $t$ is sent from host to device. In this way, CPU host controls GPU device kernels' launch, according to the minimum communication ($O(1)$) with GPU device.

## 4.3 Computational Results

In this section we report timings and speedups.

### 4.3.1 Test Problems

We choose two classes of benchmark polynomial systems that can be formulated for any dimension. In the first problem, we bootstrap from a linear system into a gradually higher dimensional and higher degree problem. Monodromy is applied in the second benchmark problem and the homotopies connect polynomial systems of the same complexity.

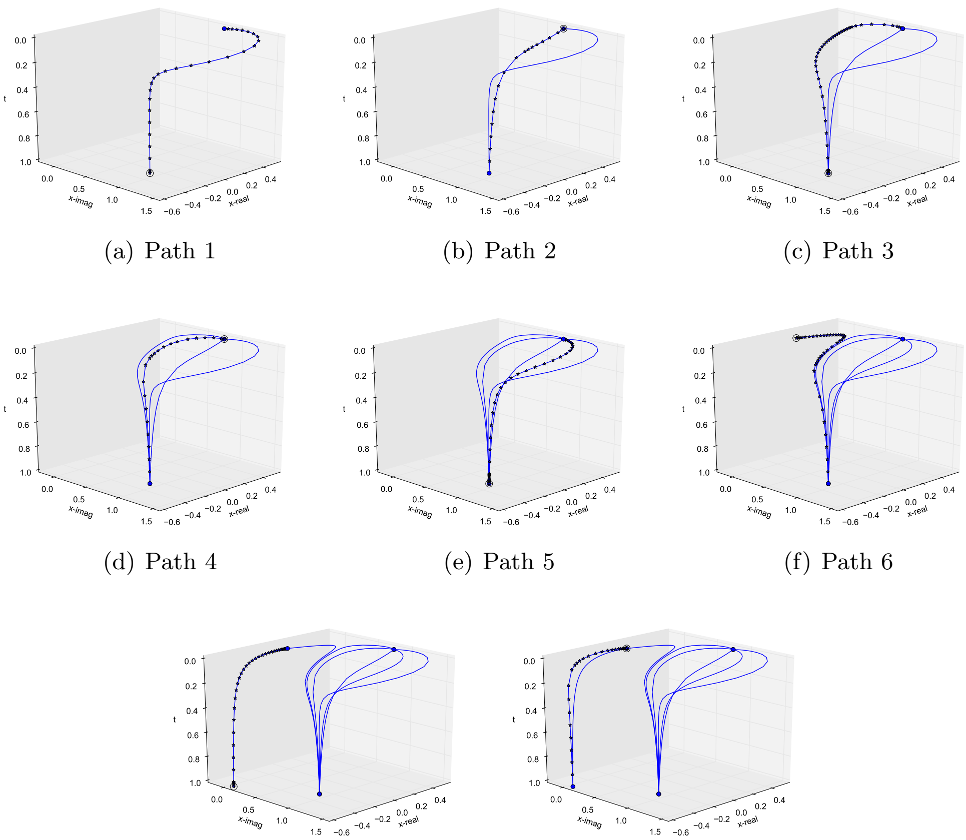
#### 4.3.1.1 Monodromy on cyclic $n$-roots

Cyclic $n$-roots is first introduced in Section 2.5.1.1. Backelin's Lemma (6) states that this system has a solution set of dimension $m-1$ for $n = \ell m^2$, where $\ell$ is no multiple of $k^2$, for $k \geq 2$. The system benchmarks polynomial solvers, see (15; 18; 47). In (3; 1) we derived an explicit parameter representation for those positive dimensional cyclic $n$-roots solution sets. To compute the degree of the sets, we add as many linear equations $\mathbf{L}$ (with random complex coefficients) as

**Algorithm 7** Accelerated tracking of one single path

**procedure** GPU_NEWTON_REFINE($Inst, W, P$)
  $t \leftarrow 0$,
  $\Delta t \leftarrow P.max\Delta t$
  $\#successes \leftarrow 0$
  $\#steps \leftarrow 0$
  **while** $t < 1$ **do**
    **if** $\#steps > P.max\#steps$ **then**
      return $fail$
    **end if**
    $t = \min(1, t + \Delta t)$
    copy $t$ from host to GPU
    launch kernel PREDICT($W.\mathbf{x}\_array$, $W.t\_array$, $W.x\_t\_idx$)
    $newton\_success$ = GPU_NEWTON_PATH($Inst$,$W$,$P$)
    **if** $newton\_success$ **then**
      Update array index $W.x\_t\_idx$
      Update pointer of $W.\mathbf{x}$ in $W.\mathbf{x}\_array$, $W.t$ in $W.t\_array$
      $\#successes = \#successes + 1$
      **if** $\#successes > 2$ **then**
        $\Delta t = \min(\Delta t * P.step\_increase, P.max\Delta t)$
      **end if**
    **else**
      $\#successes = 0$
      $\Delta t = \Delta t * P.step\_decrease$
      **if** $\Delta t < P.min\_\Delta t$ **then**
        return $fail$
      **end if**
    **end if**
    $\#steps = \#steps + 1$
  **end while**
  return $success$
**end procedure**

Figure 27: Visualization of monodromy on cyclic 4-roots




(a) Path 1 (b) Path 2 (c) Path 3

(d) Path 4 (e) Path 5 (f) Path 6

(g) Path 7 (h) Path 8

Note: cyclic 4-roots has two solution sets of degree 2. Each subfigure adds a new path. Corrected points on path are marked by a star and the target point is circled.

the dimension of the set and count the number of solutions of the system $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, augmented with $\mathbf{L}$:

$$\begin{cases} \mathbf{f}(\mathbf{x}) = \mathbf{0} \\ \mathbf{L}(\mathbf{x}) = \mathbf{0}. \end{cases} \tag{4.1}$$

The explicit representation of the cyclic $n$-roots solution sets allows for a quick calculation of the degrees, displayed in Table VII. From (2; 1, Proposition 4.2), we have that the degree $d = m$ for $n = m^2$ and this result extends for $n = \ell m^2$.

TABLE VII: Degrees $d$ of the cyclic $n$-roots solution sets.

| $n$ | 16 | 32 | 48 | 64 | 80 | 96 | 128 | 144 | 160 | 176 |
|---|---|---|---|---|---|---|---|---|---|---|
| $d$ | 4 | 4 | 4 | 8 | 4 | 4 | 8 | 12 | 4 | 4 |
| $n$ | 192 | 208 | 240 | 256 | 272 | 288 | 304 | 320 | 336 | 352 |
| $d$ | 8 | 4 | 4 | 16 | 4 | 12 | 4 | 8 | 4 | 4 |

Observe that many solution sets in Table VII have degree four. A fourth-order predictor will give accurate predictions on a surface of degree four. Therefore, the numerically harder problems are those dimensions for which the degree of the solution set is larger than four. For cyclic 64-roots double precision is no longer sufficient.

As done in PHCpack (48), with monodromy, the degree is computed numerically, using a sequence of homotopies:

$$\mathbf{h}_\alpha(\mathbf{x},t) = \begin{cases} \mathbf{f}(\mathbf{x}) = \mathbf{0} \\ \alpha(1-t)\mathbf{L}(\mathbf{x}) + t\mathbf{K}(\mathbf{x}) = \mathbf{0} \end{cases} \tag{4.2}$$

$$\mathbf{h}_\beta(\mathbf{x},t) = \begin{cases} \mathbf{f}(\mathbf{x}) = \mathbf{0} \\ \beta(1-t)\mathbf{K}(\mathbf{x}) + t\mathbf{L}(\mathbf{x}) = \mathbf{0} \end{cases} \tag{4.3}$$

where $\mathbf{K}(\mathbf{x}) = \mathbf{0}$ is as $\mathbf{L}(\mathbf{x}) = \mathbf{0}$ another set of linear equations with random coefficients and where $\alpha$ and $\beta$ are different random complex constants. One loop consists in tracking one path defined by $\mathbf{h}_\alpha(\mathbf{x},t) = \mathbf{0}$ and $\mathbf{h}_\beta(\mathbf{x},t) = \mathbf{0}$. In both cases $t$ goes from 0 to 1. See Figure 27.

After sufficiently many loops, each time for different values of the random constants $\alpha$ and $\beta$, we will find as many different solutions of the system (Equation 4.1) as the degree of the solution set, as in Table VII.

#### 4.3.1.2 Matrix completion with Pieri homotopies

Pieri hypersurface problem is first introduced in Section 2.5.1.2. In the application of Pieri homotopy algorithm (30; 31; 50), we consider matrices $X$:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & x_{3,2} \\ 0 & 0 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & x_{3,2} \\ 0 & x_{4,2} \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 \\ x_{2,1} & 1 \\ 0 & x_{3,2} \\ 0 & x_{4,2} \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 \\ x_{2,1} & 1 \\ x_{2,2} & x_{3,2} \\ 0 & x_{4,2} \end{bmatrix}, \tag{4.4}$$

and then ends in the matrix X of (Equation 2.11). Each matrix in the sequence introduces one new variable and the homotopy starts at a solution of the previous homotopy, extended with a zero value for the new variable, each time a new matrix $A$ is introduced.

Using a superscript to index a sequence of matrices, $A^{(i)} \in \mathbb{C}^{n\times m}$, $i = 1, 2, \ldots, k$, Pieri homotopies are defined as

$$\mathbf{h}(\mathbf{x}, t) = \begin{cases} \det(A^{(i)}|X) = 0, \; i = 1, 2, \ldots, k-1, \\ \det(tA^{(k)} + (1-t)S_X|X) = 0, \end{cases} \tag{4.5}$$

where $S_X$ is a special matrix which ensures that for $t = 0$, we have start solutions by setting the bottommost variables of $X$ to zero. Because of the similarities in the monomial structure, for this fully determined type of Pieri homotopy we may consider as last equation in the homotopy

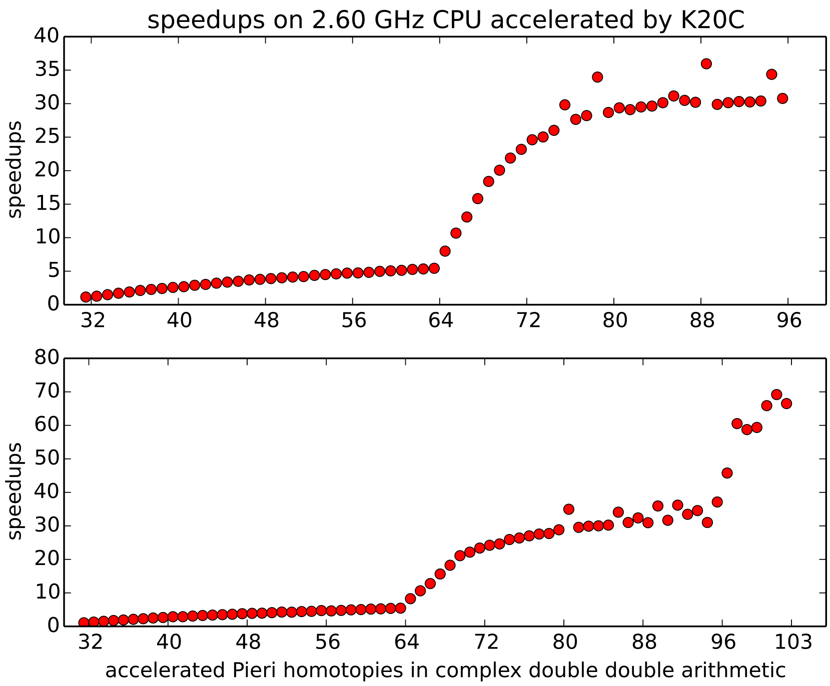
$$t \det(A^{(k)}|X) + (1-t)\det(S_X|X) = 0. \tag{4.6}$$

In the sequence of homotopies, the index $k$ runs from 1 to $m \times p$. Because in our setup, we track one single path, we may start at $k = m - 1$, which corresponds to a linear system as only the last column of $X$ contains variables. As $k$ increases, the polynomial homotopy becomes more and more nonlinear. In the last stages of the homotopy, for $p = 3$, the cost of evaluation via the minor expansions becomes cubic in $n$. As the cost of evaluation and differentiation becomes dominant, the most important factor lies in the summation of the many terms in every polynomial.

### 4.3.2 Running Pieri homotopies

Table VIII and Table IX summarize the execution of two sequences of Pieri homotopies, Table VIII is the first instance for dimensions $n$ ranging from 32 and 96. Table IX is the second instance for dimensions $n$ ranging from 32 and 103. For each path, we list the number $m$ of predictor-corrector stages, as we use the same step length control strategy.

Figure 28: The speedups of two sequences of pieri homotopies.




Note: The first instance on top visualizes Table VIII. The second instance below visualizes Table IX. Observe the different ranges of the vertical axes.

TABLE VIII: Running the first instance of Pieri homotopies in complex double double arithmetic, from dimensions 32 to 96.

| $n$ | $m$ | cpu | gpu | S | $n$ | $m$ | cpu | gpu | S |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 25 | 0.05 | 0.05 | 1.1 | 65 | 63 | 5.6 | 0.7 | 8.0 |
| 33 | 150 | 0.70 | 0.55 | 1.3 | 66 | 187 | 25.3 | 2.4 | 10.7 |
| 34 | 65 | 0.44 | 0.30 | 1.5 | 67 | 98 | 17.4 | 1.3 | 13.1 |
| 35 | 101 | 0.84 | 0.49 | 1.7 | 68 | 239 | 50.0 | 3.2 | 15.8 |
| 36 | 36 | 0.40 | 0.21 | 1.9 | 69 | 244 | 68.0 | 3.7 | 18.4 |
| 37 | 10 | 0.13 | 0.06 | 2.1 | 70 | 118 | 40.4 | 2.0 | 20.1 |
| 38 | 37 | 0.56 | 0.25 | 2.3 | 71 | 41 | 17.0 | 0.8 | 21.9 |
| 39 | 24 | 0.44 | 0.18 | 2.4 | 72 | 89 | 41.8 | 1.8 | 23.2 |
| 40 | 19 | 0.39 | 0.15 | 2.6 | 73 | 99 | 44.2 | 1.8 | 24.6 |
| 41 | 52 | 1.12 | 0.41 | 2.7 | 74 | 85 | 41.9 | 1.7 | 25.0 |
| 42 | 66 | 1.38 | 0.48 | 2.9 | 75 | 89 | 50.3 | 1.9 | 26.0 |
| 43 | 72 | 1.67 | 0.55 | 3.0 | 76 | 246 | 136.0 | 4.6 | 29.8 |
| 44 | 23 | 0.61 | 0.19 | 3.2 | 77 | 100 | 53.3 | 1.9 | 27.7 |
| 45 | 16 | 0.45 | 0.13 | 3.4 | 78 | 81 | 45.7 | 1.6 | 28.2 |
| 46 | 25 | 0.74 | 0.21 | 3.5 | 79 | 272 | 210.0 | 6.2 | 34.0 |
| 47 | 27 | 0.90 | 0.24 | 3.7 | 80 | 226 | 158.0 | 5.5 | 28.7 |
| 48 | 53 | 1.69 | 0.45 | 3.8 | 81 | 50 | 39.0 | 1.3 | 29.4 |
| 49 | 32 | 1.05 | 0.27 | 3.9 | 82 | 116 | 91.2 | 3.1 | 29.1 |
| 50 | 108 | 3.77 | 0.94 | 4.0 | 83 | 136 | 107.2 | 3.6 | 29.5 |
| 51 | 48 | 1.67 | 0.41 | 4.1 | 84 | 69 | 59.2 | 2.0 | 29.6 |
| 52 | 79 | 2.97 | 0.71 | 4.2 | 85 | 248 | 206.5 | 6.9 | 30.1 |
| 53 | 53 | 2.06 | 0.47 | 4.4 | 86 | 181 | 166.5 | 5.3 | 31.2 |
| 54 | 91 | 3.37 | 0.75 | 4.5 | 87 | 32 | 31.1 | 1.0 | 30.5 |
| 55 | 18 | 0.90 | 0.19 | 4.6 | 88 | 36 | 37.3 | 1.2 | 30.2 |
| 56 | 28 | 1.37 | 0.29 | 4.7 | 89 | 94 | 113.7 | 3.2 | 36.0 |
| 57 | 45 | 2.01 | 0.42 | 4.7 | 90 | 73 | 75.7 | 2.5 | 29.9 |
| 58 | 34 | 1.69 | 0.35 | 4.8 | 91 | 66 | 68.4 | 2.3 | 30.1 |
| 59 | 29 | 1.41 | 0.28 | 5.0 | 92 | 90 | 98.0 | 3.2 | 30.3 |
| 60 | 111 | 5.70 | 1.13 | 5.1 | 93 | 102 | 112.6 | 3.7 | 30.3 |
| 61 | 67 | 3.77 | 0.74 | 5.1 | 94 | 41 | 40.2 | 1.3 | 30.4 |
| 62 | 42 | 2.40 | 0.46 | 5.3 | 95 | 53 | 61.4 | 1.8 | 34.4 |
| 63 | 85 | 4.85 | 0.91 | 5.3 | 96 | 64 | 67.2 | 2.2 | 30.8 |
| 64 | 63 | 3.36 | 0.62 | 5.4 | | | | | |

Note: timing in seconds. The last column for each dimension contains the speedup S.

TABLE IX: Running the second instance of Pieri homotopies in complex double double arithmetic, from dimensions 32 to 103.

| $n$ | $m$ | cpu | gpu | S | $n$ | $m$ | cpu | gpu | S |
|---|---|---|---|---|---|---|---|---|---|
| 32 | 16 | 0.03 | 0.03 | 1.1 | 68 | 51 | 11.3 | 0.7 | 15.6 |
| 33 | 68 | 0.38 | 0.30 | 1.3 | 69 | 60 | 17.9 | 1.0 | 18.2 |
| 34 | 21 | 0.16 | 0.10 | 1.5 | 70 | 22 | 7.7 | 0.4 | 21.1 |
| 35 | 19 | 0.20 | 0.11 | 1.7 | 71 | 156 | 62.0 | 2.8 | 22.1 |
| 36 | 18 | 0.20 | 0.11 | 1.9 | 72 | 39 | 19.4 | 0.8 | 23.4 |
| 37 | 34 | 0.44 | 0.21 | 2.1 | 73 | 49 | 26.9 | 1.1 | 24.2 |
| 38 | 32 | 0.44 | 0.19 | 2.3 | 74 | 98 | 56.7 | 2.3 | 24.6 |
| 39 | 41 | 0.65 | 0.26 | 2.5 | 75 | 74 | 43.6 | 1.7 | 25.9 |
| 40 | 12 | 0.25 | 0.09 | 2.6 | 76 | 63 | 38.8 | 1.5 | 26.4 |
| 41 | 17 | 0.36 | 0.13 | 2.9 | 77 | 37 | 27.1 | 1.0 | 27.0 |
| 42 | 29 | 0.65 | 0.23 | 2.9 | 78 | 95 | 67.9 | 2.5 | 27.6 |
| 43 | 66 | 1.47 | 0.48 | 3.1 | 79 | 112 | 76.6 | 2.8 | 27.7 |
| 44 | 10 | 0.28 | 0.08 | 3.2 | 80 | 157 | 115.0 | 4.0 | 28.8 |
| 45 | 46 | 1.30 | 0.39 | 3.4 | 81 | 321 | 265.2 | 7.6 | 35.0 |
| 46 | 31 | 0.85 | 0.24 | 3.5 | 82 | 63 | 47.9 | 1.6 | 29.6 |
| 47 | 51 | 1.60 | 0.44 | 3.6 | 83 | 42 | 33.3 | 1.1 | 29.9 |
| 48 | 16 | 0.54 | 0.14 | 3.8 | 84 | 19 | 17.3 | 0.6 | 30.0 |
| 49 | 16 | 0.58 | 0.15 | 3.9 | 85 | 224 | 188.1 | 6.2 | 30.2 |
| 50 | 24 | 0.91 | 0.23 | 3.9 | 86 | 159 | 147.9 | 4.3 | 34.1 |
| 51 | 62 | 2.31 | 0.56 | 4.1 | 87 | 252 | 199.4 | 6.4 | 31.0 |
| 52 | 40 | 1.52 | 0.36 | 4.3 | 88 | 574 | 431.2 | 13.3 | 32.4 |
| 53 | 46 | 2.09 | 0.49 | 4.2 | 89 | 213 | 171.3 | 5.5 | 30.9 |
| 54 | 33 | 1.62 | 0.37 | 4.4 | 90 | 137 | 129.9 | 3.6 | 35.9 |
| 55 | 79 | 3.84 | 0.86 | 4.5 | 91 | 187 | 157.8 | 5.0 | 31.7 |
| 56 | 36 | 1.71 | 0.36 | 4.7 | 92 | 250 | 219.8 | 6.1 | 36.2 |
| 57 | 42 | 2.23 | 0.48 | 4.6 | 93 | 847 | 646.1 | 19.3 | 33.4 |
| 58 | 29 | 1.58 | 0.33 | 4.8 | 94 | 199 | 169.1 | 4.9 | 34.6 |
| 59 | 37 | 1.98 | 0.40 | 4.9 | 95 | 108 | 96.1 | 3.1 | 31.0 |
| 60 | 16 | 0.95 | 0.19 | 5.0 | 96 | 190 | 230.8 | 6.2 | 37.1 |
| 61 | 37 | 2.10 | 0.41 | 5.2 | 97 | 161 | 305.3 | 6.7 | 45.8 |
| 62 | 50 | 2.97 | 0.57 | 5.2 | 98 | 76 | 264.7 | 4.4 | 60.5 |
| 63 | 34 | 1.95 | 0.36 | 5.4 | 99 | 75 | 322.6 | 5.5 | 58.7 |
| 64 | 75 | 4.54 | 0.84 | 5.4 | 100 | 242 | 1367.2 | 23.0 | 59.4 |
| 65 | 83 | 7.93 | 0.96 | 8.3 | 101 | 809 | 4655.7 | 70.7 | 65.9 |
| 66 | 195 | 27.20 | 2.56 | 10.6 | 102 | 1016 | 5231.7 | 75.6 | 69.2 |
| 67 | 154 | 26.43 | 2.07 | 12.8 | 103 | 375 | 2923.2 | 44.0 | 66.5 |

Note: timing in seconds. The last column for each dimension contains the speedup S.

Because the fluctuations in the number of predictor-corrector steps along a path can vary by a factor as large as five, the single digit speedups obtained by acceleration in low dimensions is often in the same range as the factor in the fluctuations of the timings. While fluctuations in larger dimensions remain of the same order, the double digit speedups make that with acceleration we may increase the dimension, compare for example the lines for $n = 63$ and $n = 96$ in Table VIII and still be faster: 2.2 seconds versus 4.85 seconds.

Double digit speedups arise after dimension 65. After dimension 97, the speedup then almost doubles, see Figure 28.

### 4.3.3 Running one path of cyclic $n$-roots

Table X summarizes the computational results from running one path on a homotopy to apply the monodromy on the cyclic $n$-roots problem. The first case where double precision does not suffice is in dimension $n = 64$, but the path can then be tracked successfully in double double precision. For $n = 144$, both double and double double precision are insufficient and quad double precision is needed.

The difficulties could be explained by the higher degree of the solution set. Cyclic 256-roots remains a challenge. The double digit speedups obtained by acceleration implies that we can offset the cost of one extra level of higher precision.

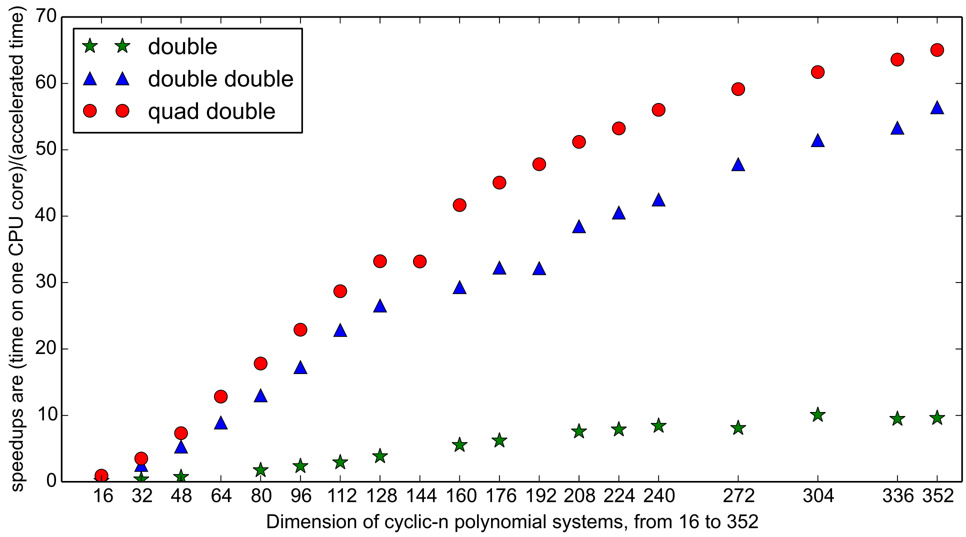
The data in Table X for complex double and complex double double precision is visualized in Figure 29. Concerning the data in Table X, let us compare the accelerated times in double double precision to the times on one CPU core in double precision. For the last line, observe that it takes 93.89 seconds to track one path in double precision without acceleration. With

TABLE X: Running one path of cyclic $n$-roots in various precisions and in various dimensions.

| | complex double | | | | complex double double | | | | complex quad double | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | $m$ | cpu | gpu | S | $m$ | cpu | gpu | S | $m$ | cpu | gpu | S |
| 16 | 32 | 0.00 | 0.03 | 0.14 | 20 | 0.04 | 0.06 | 0.65 | 20 | 0.48 | 0.52 | 0.92 |
| 32 | 100 | 0.06 | 0.16 | 0.35 | 79 | 1.03 | 0.41 | 2.53 | 79 | 12.66 | 3.62 | 3.50 |
| 48 | 103 | 0.17 | 0.24 | 0.72 | 78 | 3.23 | 0.61 | 5.29 | 78 | 39.46 | 5.39 | 7.32 |
| 64 | 0 | | | | 225 | 22.94 | 2.57 | 8.92 | 181 | 229.99 | 17.93 | 12.83 |
| 80 | 99 | 0.73 | 0.42 | 1.74 | 75 | 14.96 | 1.15 | 13.01 | 75 | 180.37 | 10.13 | 17.81 |
| 96 | 95 | 1.23 | 0.52 | 2.36 | 69 | 23.17 | 1.34 | 17.26 | 69 | 289.38 | 12.64 | 22.90 |
| 112 | 171 | 3.42 | 1.17 | 2.92 | 121 | 68.07 | 2.98 | 22.86 | 121 | 813.91 | 28.36 | 28.70 |
| 128 | 162 | 5.66 | 1.47 | 3.85 | 123 | 102.94 | 3.88 | 26.54 | 123 | 1253.82 | 37.75 | 33.21 |
| 144 | 0 | | | | 0 | | | | 1074 | 15898.67 | 479.18 | 33.18 |
| 160 | 68 | 4.84 | 0.87 | 5.53 | 49 | 83.11 | 2.84 | 29.31 | 49 | 998.43 | 23.96 | 41.67 |
| 176 | 160 | 15.65 | 2.52 | 6.21 | 118 | 259.80 | 8.06 | 32.24 | 118 | 3179.81 | 70.58 | 45.05 |
| 192 | 0 | | | | 150 | 419.16 | 13.03 | 32.16 | 143 | 5054.70 | 105.69 | 47.83 |
| 208 | 231 | 39.51 | 5.22 | 7.57 | 168 | 628.46 | 16.33 | 38.48 | 168 | 7529.02 | 147.09 | 51.19 |
| 224 | 96 | 19.39 | 2.46 | 7.88 | 71 | 319.27 | 7.88 | 40.54 | 71 | 3925.33 | 73.76 | 53.22 |
| 240 | 140 | 34.04 | 4.04 | 8.42 | 96 | 531.01 | 12.49 | 42.50 | 96 | 6714.01 | 119.86 | 56.01 |
| 256 | 0 | | | | 0 | | | | 0 | | | |
| 272 | 160 | 58.19 | 7.19 | 8.09 | 118 | 914.24 | 19.12 | 47.82 | 118 | 10829.36 | 183.12 | 59.14 |
| 288 | 0 | | | | 0 | | | | 0 | | | |
| 304 | 142 | 81.04 | 8.05 | 10.07 | 103 | 1176.29 | 22.87 | 51.44 | 103 | 13992.60 | 226.78 | 61.70 |
| 320 | 0 | | | | 0 | | | | 0 | | | |
| 336 | 157 | 105.30 | 11.12 | 9.47 | 114 | 1772.97 | 33.26 | 53.31 | 114 | 20807.27 | 327.25 | 63.58 |
| 352 | 121 | 93.89 | 9.78 | 9.60 | 90 | 1621.15 | 28.75 | 56.39 | 90 | 18881.13 | 290.36 | 65.03 |

Note: Data in the column under the header $m$ indicates the number of predictor-corrector steps. If the path fails, $m = 0$. The last column for each dimension and precision contains the speedup S. Timing in seconds.

Figure 29: Running one path of cyclic $n$-roots in various precisions and in various dimensions.




acceleration tracking one path in double double precision takes 28.75 seconds, so we can double the precision and still be three times faster than in double precision without acceleration. Speedups computed in Table X are shown in Figure 29. We see that in double double precision, the speedups rise faster as the dimension increases than in double precision.

### 4.3.4 Conclusion

With many sequential steps of predictions and corrections, the CPU host controls the GPU device by the minimum feedback. Both speed up and quality up are achieved by the GPU's acceleration. Further work includes automatic determination of the required level of precision and the multi-precision path tracking.

# CHAPTER 5

# MULTIPLE PATH TRACKING ON GPUS

To find all isolated solutions, we need to track multiple solutions paths. Given a polynomial system with multiple start solutions $\mathbf{x}_i(0)$, all paths $\mathbf{x}_i(t)$ are tracked independently to the target solutions $\mathbf{x}_i(1)$.

With Single instruction, multiple thread (SIMT) model, we want GPU device to track multiple paths. The challenge is that paths need different number of steps, and also, correctors of paths need different number of Newton's iterations. Thus, we need a unified pattern to combine all paths, in order to build a SIMT model. The ideas in this chapter are presented in our paper (60).

## 5.1 SIMT multiple Path Tracking

From analysis in Section 4.2, the path tracking has three basic compute-intensive parts, predictor, polynomial evaluation and differentiation (PED) and Modified Gram-Schmidt (MGS). First, we synchronize all paths to work on the same parts like 31(a).

For each stage, each job is associated with its *path_idx*. The number total of jobs N_Job indicates grid sizes for GPU kernels, so GPU threads locate their own jobs by *path_idx*. See 31(b).

The number of jobs N_job and the array of *path_idx* can be generated in parallel on GPUs. After each stage, there is a check kernel to determine the status of all paths. There are three status in Newton's method: 0 is to continue, $-1$ is for failure or 1 is for success. Then all

Figure 30: Simplified SIMT of one predictor-corrector step on three paths.

| Path0 | Path1 | Path2 |
|---|---|---|
| PREDICT | PREDICT | PREDICT |
| PED<br>MGS | PED<br>MGS | PED<br>MGS |
| PED<br>MGS | | PED<br>MGS |
| | | PED<br>MGS |

(a) Unified pattern

$\Longrightarrow$

| Job_0 | Job_1 | Job_2 | N_Job |
|---|---|---|---|
| PREDICT_0 | PREDICT_1 | PREDICT_2 | 3 |
| PED_0<br>MGS_0 | PED_1<br>MGS_1 | PED_2<br>MGS_2 | 3<br>3 |
| PED_0<br>MGS_0 | PED_2<br>MGS_2 | | 2<br>2 |
| PED_2<br>MGS_2 | | | 1<br>1 |

(b) Unified pattern with job indices

Note: The first path needs two Newton's iterations, the second path needs only one, and the third path needs three.

Figure 31: Generated the job array of *path_idx* from current iteration status for the next stage

| | path0 | path1 | path2 | path3 | path4 | $\cdots$ |
|---|---|---|---|---|---|---|
| path status | 0 | 1 | 0 | $-1$ | 0 | $\cdots$ |
| scan for 0 | 1 | 1 | 2 | 2 | 3 | $\cdots$ |
| $job\_idx + 1$ | 1 | | 2 | | 3 | $\cdots$ |
| $path\_idx$ | 0 | | 2 | | 4 | $\cdots$ |

paths with status 0's are counted by a parallel scan(prefix sum). In the scan array, element of *path_idx* with status 0's is $job\_idx + 1$. Thus, we can generate the array of *path_idx* for the next stage. The last element of the scan array is the number of jobs N_job. See Figure 31.

## 5.2 Newton's method for multiple path tracking on GPUs

Based on SIMT, we generalize the Algorithm 5 in Section 3.3.2 for multiple paths. $\parallel \mathbf{f}(\mathbf{x}_k) \parallel$ and its ratio to $\parallel x_k \parallel$, $\parallel \mathbf{f}(\mathbf{x}_k) \parallel > \parallel \mathbf{f}(\mathbf{x}_{k-1}) \parallel$ are the standards to check the convergence of one Newton iteration. When we track one single path, these standards are checked by the host. For multiple paths, all paths has theirs own conditions, with $O(n\_path)$ complexity, Thus we check them on the device and generate the array of *path_idx*, in order to minimize communication between host and device. After each check point, only one integer *n_path* is copied from device.

**Algorithm 8** An accelerated Newton's method for tracking multiple paths

**procedure** GPU_NEWTON_PATH_MULT($Inst, W, P$)<br>
    **for** $k$ from 1 to $P.max\_iteration$ **do**<br>
        GPU_PED_MULT($Inst$,$W$)<br>
        launch kernel MAX_ARRAY_MULT($W.matrix\_vertical$, $W.max\_eq\_val$)<br>
        launch kernel MAX_ARRAY_MULT($W.\mathbf{x}\_vertical$, $W.max\_\mathbf{x}$)<br>
        launch kernel CHECK_PED($W.max\_eq\_val$, $W.max\_\mathbf{x}$, $W.Newton\_Success$)<br>
        launch kernel CHECK_PATH_IDX($W.success$, $W.n\_success$, $W.path\_idx$, $W.n\_path$)<br>
        copy $n\_path$ from device to host<br>
        **if** $n\_path = 0$ **then**<br>
            break<br>
        **end if**<br>
        GPU_MGS_MULT($W$)<br>
        launch kernel MAX_ARRAY_MULT($W.\Delta\mathbf{x}$, $W.max\_\Delta\mathbf{x}$)<br>
        launch kernel UPDATE_**x**_MULT($W.\mathbf{x}$,$W.\Delta\mathbf{x}$)<br>
        launch kernel CHECK_MGS($W.max\_\Delta\mathbf{x}$, $W.max\_\mathbf{x}$, $W.Newton\_Success$)<br>
        launch kernel CHECK_PATH_IDX($W.success$, $W.n\_success$, $W.path\_idx$, $W.n\_path$)<br>
        copy $n\_path$ from device to host<br>
        **if** $n\_path = 0$ **then**<br>
            break<br>
        **end if**<br>
    **end for**<br>
**end procedure**

## 5.3 Multiple path tracking on GPUs

Similar like Newton's method, we generalize Algorithm7 in Section 4.2 for multiple paths. When we track one single path, the step size control can be performed by the host. When tracking many solution paths, every solution path has its own continuation parameter $t$ and step size $\Delta t$, with $O(n\_path)$ complexity. To minimize communication between CPU and GPU, the step size control is executed on the device. After each check point, only one integer $n\_path$ is copied from device.

**Algorithm 9** Accelerated tracking of multiple paths

**procedure** GPU_NEWTON_PATH_MULT($Inst, W, P$)
  launch kernel PATH_INIT($W.\mathbf{x}\_array$, $W.t\_array$)
  **while** *true* **do**
    launch kernel PREDICT_MULT($W.\mathbf{x}\_array$, $W.t\_array$)
    GPU_NEWTON_MULT($Inst$,$W$,$P$)
    launch kernel STEP_CONTROL($W.t$, $W.\Delta t$, $W.success$, $W.n\_success$, $P.step\_increase$, $P.step\_decrease$)
    launch kernel CHECK_PATH_IDX($W.success$, $W.n\_success$, $P.path\_idx$, $W.n\_path$)
    copy $n\_path$ from device to host
    **if** $n\_path = 0$ **then**
      break
    **end if**
  **end while**
  copy $W.success$ from device to host
  return *success*
**end procedure**

## 5.4 Computational Results

For testing multiple path tracking, we choose polynomial systems of relative small dimension, like the multiple PEDs in Section 2.5.3.

Results for tracking many paths for the cyclic 10-roots problem are summarized in Table XI. Observe the quality up. Tracking 10,000 paths in double double arithmetic takes 10 seconds on GPUs, while on the CPU it takes 26.562 seconds in double arithmetic. With our accelerated code we obtain solutions in a precision that is twice as large in a time that is more than twice as fast.

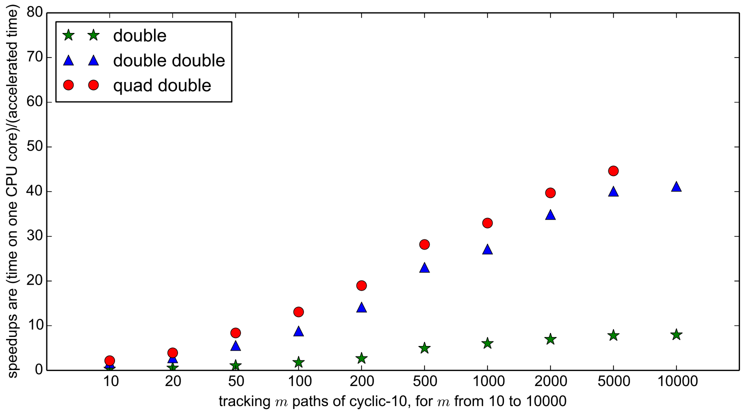
Table III lists times and speedups for evaluating and differentiating the Nash equilibrium system. Times for path tracking are listed in Table XII. Table IV lists times and speedups for evaluating and differentiating the Pieri hypersurface system. Times for path tracking are listed in Table XIII. Table XII lists times and speedups for tracking many paths of the Nash equilibrium system.

In Figure 32 we visualize these data. Notice that, as the Nash equilibrium system has more monomials than the cyclic 10-roots system, the speedups for `nash8` are better than those form `cyclic10`. The speedups improve slightly for the Pieri problem, but with a larger of number of monomials the memory allows for fewer paths to be tracked simultaneously.

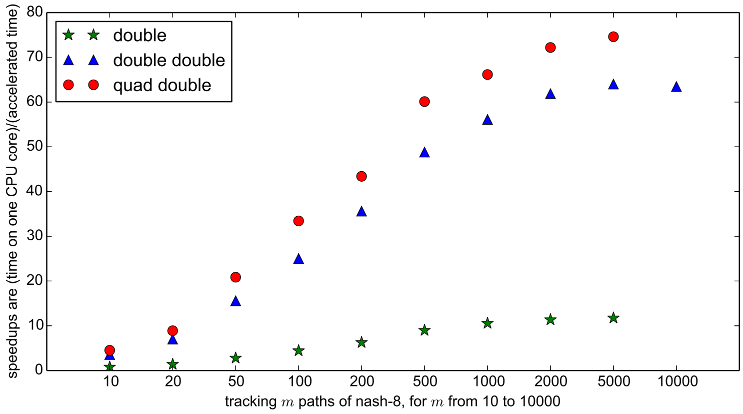
### 5.4.1 Conclusion

With the number of solution paths in polynomial homotopies reaches several hundreds, acceleration with GPUs achieves both speed up and quality up, even for polynomial homotopies of small dimension. Future work includes multicore parallelism for multiple CPUs and GPUs.

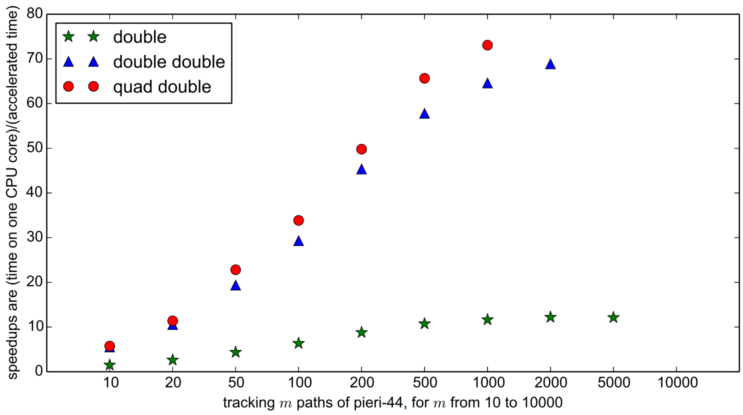
Figure 32: Speedups for tracking many paths of three polynomial systems




(a) cyclic 10-roots




(b) Nash equilibrium system




(c) Pieri hypersurface system

Note: All of them are computed in complex double, double double, and quad double arithmetic.

TABLE XI: Speedups for tracking a number of paths of the cyclic 10-roots system

complex double arithmetic

| #paths | CPU | GPU | speedup |
|---:|---:|---:|---:|
| 10 | 0.040 | 0.128 | 0.31 |
| 20 | 0.075 | 0.139 | 0.54 |
| 50 | 0.158 | 0.147 | 1.07 |
| 100 | 0.277 | 0.155 | 1.79 |
| 200 | 0.482 | 0.181 | 2.67 |
| 500 | 1.239 | 0.250 | 4.96 |
| 1000 | 2.609 | 0.432 | 6.03 |
| 2000 | 5.341 | 0.768 | 6.96 |
| 5000 | 13.358 | 1.711 | 7.81 |
| 10000 | 26.562 | 3.334 | 7.97 |

complex double double arithmetic

| #paths | CPU | GPU | speedup |
|---:|---:|---:|---:|
| 10 | 0.563 | 0.344 | 1.63 |
| 20 | 1.082 | 0.386 | 2.80 |
| 50 | 2.248 | 0.404 | 5.56 |
| 100 | 3.706 | 0.421 | 8.81 |
| 200 | 6.480 | 0.458 | 14.15 |
| 500 | 16.802 | 0.729 | 23.05 |
| 1000 | 35.683 | 1.315 | 27.14 |
| 2000 | 83.601 | 2.397 | 34.87 |
| 5000 | 210.287 | 5.246 | 40.09 |
| 10000 | 414.332 | 10.063 | 41.18 |

complex quad double arithmetic

| #paths | CPU | GPU | speedup |
|---:|---:|---:|---:|
| 10 | 5.859 | 2.696 | 2.17 |
| 20 | 11.189 | 2.852 | 3.92 |
| 50 | 24.018 | 2.866 | 8.38 |
| 100 | 38.782 | 2.966 | 13.08 |
| 200 | 67.703 | 3.568 | 18.97 |
| 500 | 174.769 | 6.203 | 28.17 |
| 1000 | 368.449 | 11.175 | 32.97 |
| 2000 | 851.255 | 21.432 | 39.72 |
| 5000 | 2164.485 | 48.495 | 44.63 |

Note: timing in seconds.

TABLE XII: Speedups for tracking a number of paths of the Nash equilibrium system

complex double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 0.152 | 0.196 | 0.77 |
| 20 | 0.330 | 0.239 | 1.38 |
| 50 | 0.815 | 0.292 | 2.79 |
| 100 | 1.512 | 0.341 | 4.43 |
| 200 | 2.894 | 0.462 | 6.26 |
| 500 | 7.257 | 0.809 | 8.97 |
| 1000 | 14.171 | 1.343 | 10.55 |
| 2000 | 28.524 | 2.514 | 11.35 |
| 5000 | 72.292 | 6.156 | 11.74 |

complex double double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 2.130 | 0.595 | 3.58 |
| 20 | 4.496 | 0.641 | 7.01 |
| 50 | 11.215 | 0.720 | 15.59 |
| 100 | 20.813 | 0.831 | 25.04 |
| 200 | 40.018 | 1.124 | 35.62 |
| 500 | 100.446 | 2.057 | 48.82 |
| 1000 | 194.243 | 3.462 | 56.11 |
| 2000 | 392.615 | 6.345 | 61.87 |
| 5000 | 992.708 | 15.504 | 64.03 |

complex quad double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 20.745 | 4.593 | 4.52 |
| 20 | 42.969 | 4.835 | 8.89 |
| 50 | 106.348 | 5.101 | 20.85 |
| 100 | 198.098 | 5.926 | 33.43 |
| 200 | 383.885 | 8.846 | 43.40 |
| 500 | 986.145 | 16.407 | 60.10 |
| 1000 | 1876.226 | 28.365 | 66.15 |
| 2000 | 3805.213 | 52.710 | 72.19 |
| 5000 | 9618.930 | 128.948 | 74.60 |

Note: timing in seconds.

TABLE XIII: Speedups for tracking a number of paths of the Pieri hypersurface system

complex double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 0.757 | 0.506 | 1.50 |
| 20 | 1.580 | 0.603 | 2.62 |
| 50 | 3.883 | 0.890 | 4.36 |
| 100 | 7.800 | 1.229 | 6.35 |
| 200 | 15.813 | 1.801 | 8.78 |
| 500 | 39.861 | 3.713 | 10.74 |
| 1000 | 80.347 | 6.898 | 11.65 |
| 2000 | 161.498 | 13.232 | 12.21 |
| 5000 | 401.001 | 33.050 | 12.13 |

complex double double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 11.307 | 2.042 | 5.54 |
| 20 | 23.558 | 2.231 | 10.56 |
| 50 | 58.339 | 3.010 | 19.38 |
| 100 | 113.878 | 3.883 | 29.32 |
| 200 | 232.249 | 5.120 | 45.36 |
| 500 | 586.282 | 10.141 | 57.81 |
| 1000 | 1183.342 | 18.317 | 64.60 |
| 2000 | 2376.400 | 34.497 | 68.89 |

complex quad double arithmetic

| #paths | CPU | GPU | speedup |
|---|---|---|---|
| 10 | 111.498 | 19.403 | 5.75 |
| 20 | 234.984 | 20.642 | 11.38 |
| 50 | 583.908 | 25.590 | 22.82 |
| 100 | 1168.055 | 34.496 | 33.86 |
| 200 | 2375.275 | 47.696 | 49.80 |
| 500 | 5986.772 | 91.191 | 65.65 |
| 1000 | 12075.740 | 165.244 | 73.08 |

Note: timing in seconds.

# CHAPTER 6

# PHC WEB INTERFACE

The high speed internet and various types of user devices, like tablets and phones, inspire us to create cloud computing service for PHCpack. The ideas in this chapter are presented in our paper (8). The advantages of PHC web interface for users include:

1. No software installation is required for the user.
2. Faster computation is hosted by our computational workstation.
3. Any device from computers, cell phones to tablets has access to PHC web interface.
4. Easy graphic user interface enables the user to solve and manage polynomial systems.

The first verson of PHC Web Interface includes basic functions of solving polynomial systems from PHCpack:

1. phc -b: the black box solver in PHCpack use polyheral homotopy to the solve start system $\mathbf{g}(\mathbf{x})$ that has as many roots as mixed volume.
2. phc -p: tracking paths defined by a homotopy in one parameter.

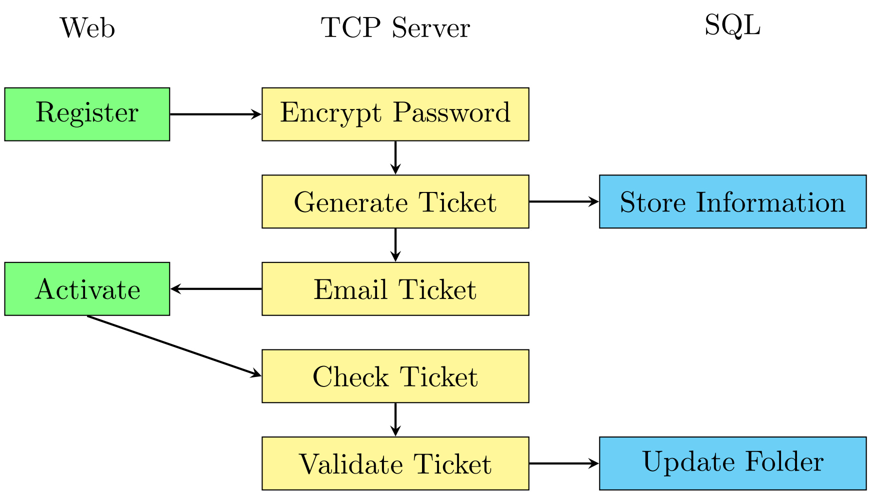
## 6.1 PHC Web Interface design

PHC Web Interface is built to server many users intuitively. Front end is a web interface. Back end includes a TCP server for job distribution, local and remote solvers, and a SQL database for user and file management. The TCP server is a process running to handle all requests from the web interface. Besides, the TCP server keeps connection to the SQL database in order to avoid extra time for SQL authorization.

### 6.1.1 Registration and activation

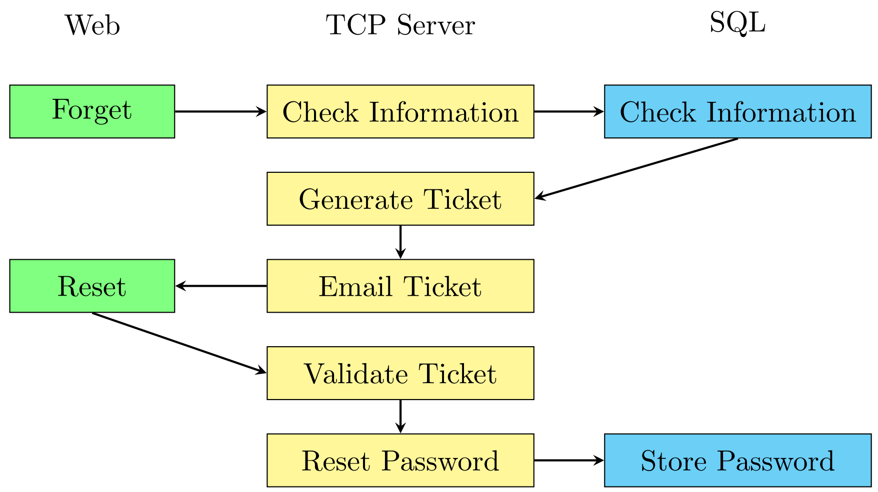
Registration and activation follow the standard process of a usual website. In this process, the Email address is used to validate a real user. See Figure 33.

Figure 33: Registration and activation process of PHC Web Interface




When a user registers, his/her information is sent to the TCP server. The TCP server encrypts the password, generates a random ticket, and store the information with password and ticket to the SQL database. After this, it sends a hyperlink containing the random ticket to the user by email. When the user clicks this hyperlink, the Web interface sends the request of activating his/her account to the TCP server. After the ticket in the request is validated by the TCP server, its creates the Folder and store it in the SQL database.

Figure 34: Resetting the password of PHC Web Interface

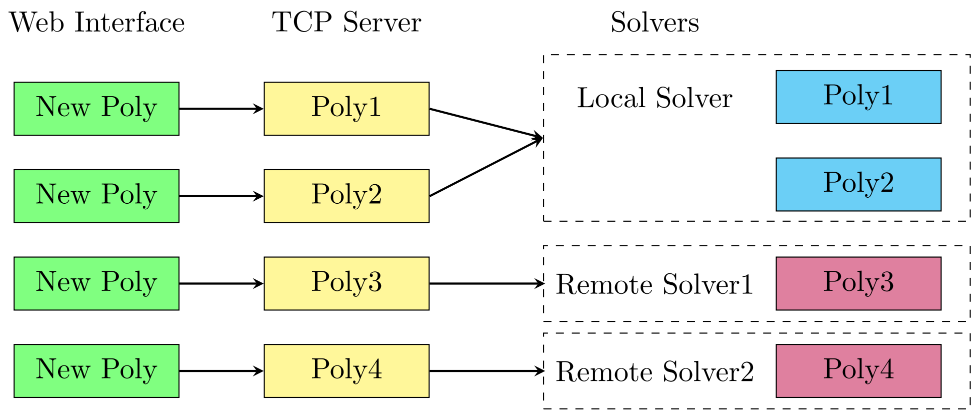



When the user forgets the password, he/she can use the information of email, name and organization to reset the password. The Web interface sends these requests to the TCP server. After the TCP server validates these informations in the SQL database, the TCP server send a new ticket for the user to reset a password by email. See Figure 34.

### 6.1.2 Job distribution based on TCP Server

The TCP server is also in charge of distributing jobs to local and remote solvers. The Web interface submits jobs of users' new polynomial systems. The TCP server is a single-thread process to put these jobs into a job queue. The TCP server first assigns these jobs to the local solver. If the local solver is fully occupied, remote solvers help solving the jobs. The remote solvers check the job queue in TCP server once in each certain time interval. If a remote solver finds a new job, it requests the file with the polynomial system from TCP server and send the solution file back after solving the job. In sum, the job distribution makes the TCP server handle many jobs simultaneously and expendable to remote computation resources.

Figure 35: Job distribution of PHC Web Interface



### 6.1.3 User management by SQL

A SQL database is constructed for PHC Web Interface. It includes two basic tables. One is the users table with the registration information of users. See 37(a). The other is the polynomial table storing basic information of all polynomial systems and solving status. See 37(b).

Figure 36: Database structure of tables in PHC Web Interface

| Name | Datatype | Description |
|---|---|---|
| $Uid$ | INT | Unique user ID |
| $Name\_First$ | CHAR(20) | First name |
| $Name\_Last$ | CHAR(20) | Last name |
| $Email$ | CHAR(40) | Email address |
| $Org$ | CHAR(40) | Organization |
| $passwd$ | CHAR(40) | Encrypted password by SHA |
| $Reg\_Date$ | DATE | Users' registration date |
| $Ticket$ | CHAR(40) | A SHA ticket for user to activate and reset password |
| $Folder$ | CHAR(40) | Name of the user's folder |
| $Status$ | SMALLINT | Status of solving the user's polynomial system |

(a) Table of users

| Name | Datatype | Description |
|---|---|---|
| $Polyid$ | INT | Unique polynomial system ID |
| $Uid$ | INT | User ID |
| $Name$ | Char(50) | Name of polynomial system |
| $Dim$ | INT | Dimension of polynomial system |
| $CTIME$ | DATETIME | Create time of polynomial system |
| $Status$ | INT | Status of solving |
| $Sols$ | INT | Number of solutions |
| $Time$ | FLOAT | Solving time |

(b) Table of polynomial systems

Each user has a randomly named folder, which stores solved polynomials, solutions and PHCpack reports. From the web interface, the user has ability to view all his/her polynomial systems, edit the names of the systems, delete the systems and check the solving status of the current polynomial system. See Figure 37.

Figure 37: Manage polynomial systems in PHC Web interface

**My Polynomial Systems**

| Name | Status | Dim | Sols | Creation Time | Solving Time | PHC Report | Actions |
|---|---|---|---|---|---|---|---|
| cyclic7-hom | Solved | 7 | 924 | 2013-08-03 05:40:57 | 8.301 | cyclic7-hom.phc | Delete |
| cyclic7 | Solved | 7 | 924 | 2013-08-03 05:39:34 | 16.381 | cyclic7.phc | Delete |
| cyclic6-hom | Solved | 6 | 156 | 2013-08-03 05:39:13 | 0.756 | cyclic6-hom.phc | Delete |
| cyclic6 | Solved | 6 | 156 | 2013-08-03 05:38:53 | 1.526 | cyclic6.phc | Delete |
| quadfor2-hom | Solved | 4 | 2 | 2013-08-03 05:21:28 | 0.008 | quadfor2-hom.phc | Delete |
| quadfor2 | Solved | 4 | 2 | 2013-08-03 05:17:47 | 0.015 | quadfor2.phc | Delete |

## 6.2 Development Environment and tools

The web interface is running in Red Hat on Microway RHEL workstation with two Intel Xeon E5-2670, 16 cores at 2.6 Ghz. Our web server is built on Apache, and Python CGI is the script language for Web interface. We use MySQL for user and data management and TCP server for job distribution. For the security of our users, we use HTTPS/SSL to encrypt the data of web page and store users' passwords encrypted SHA.

# CHAPTER 7

# A POLYNOMIAL SYSTEM DATABASE

A polynomial system can be written in different ways by permuting variables, monomials and equations. Traditional databases use text to store the polynomial systems. But based on text, it is hard to search a polynomial system from the database. The difficulty is to represent a polynomial system as a unique string.

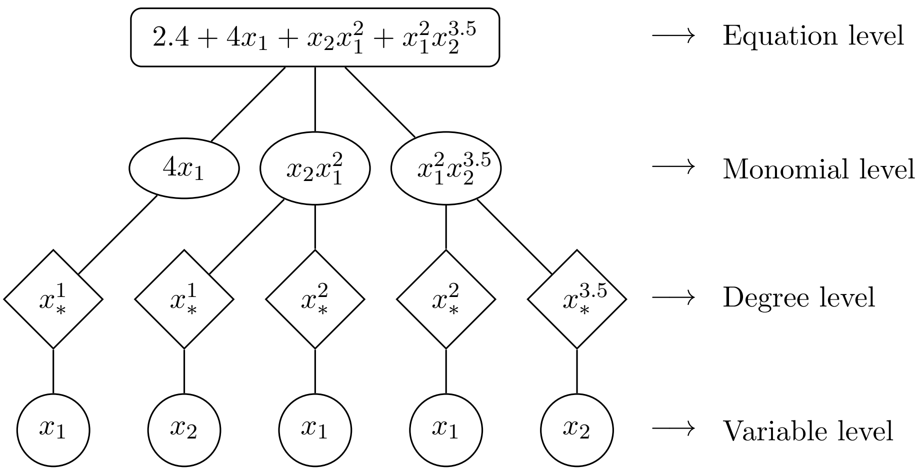
**Definition 1.** We say two polynomial systems $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are *isomorphic*, if

1. $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ have the same dimension and the same number of equations,
2. there exists a permutation $\sigma_1$ for their variables and a permutation $\sigma_2$ for their equations, such that each equation of two polynomial systems are exactly the same,

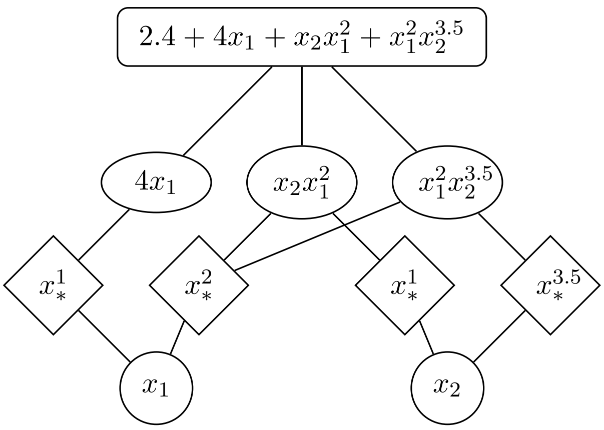
$$\mathbf{f}(\mathbf{x}) = \sigma_2(\mathbf{g})(\sigma_1(\mathbf{x})) \tag{7.1}$$

In this chapter, we develop a new approach to check the isomorphism of polynomial systems by graph. Similar ideas are presented in our paper (8). Here are some related work. Algorithms in multivariate cryptology (45) apply Grbner basis algorithms (19) and graph-theoretic algorithms (9).

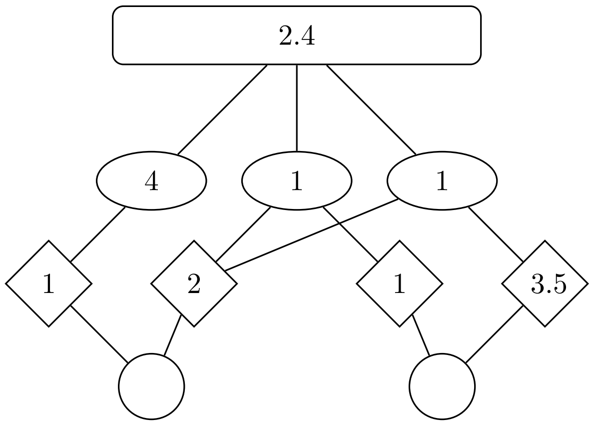
Figure 38: Generate a unique graph representation for a polynomial equation




(a) Expand and simplify the equation. Record it on a 4-level tree graph



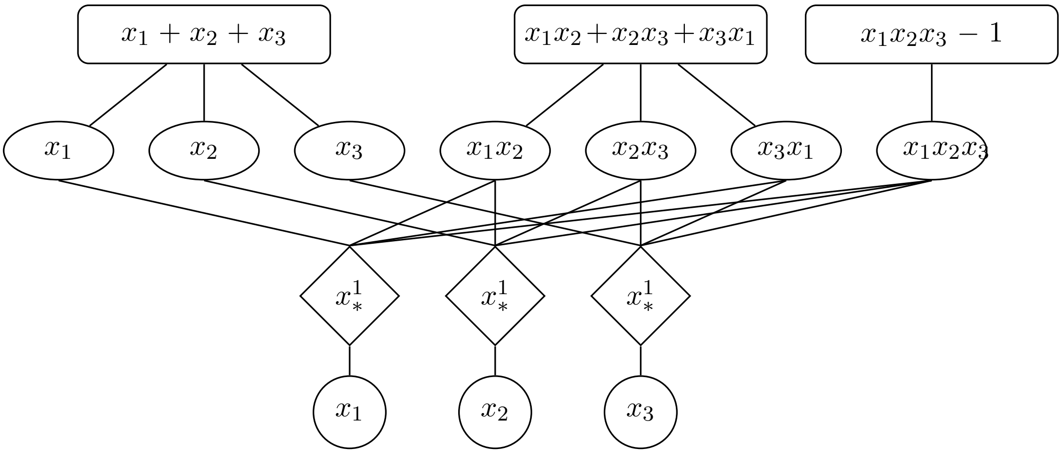

(b) Merge variable nodes and degree nodes



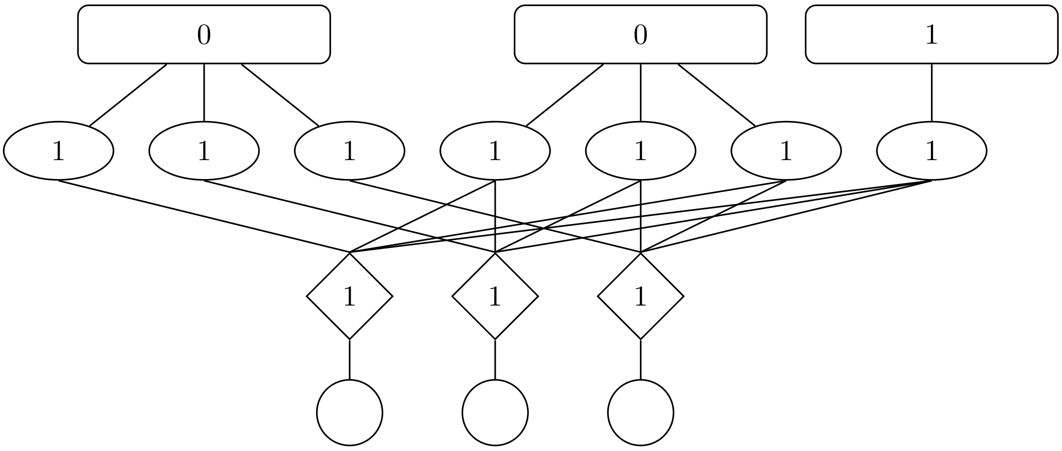

(c) Label nodes by numbers

Note: the example is $2.4+4x_1+x_2x_1^2+x_1^2x_2^{3.5}$. Type of nodes are represented as different shape: circle is for variable, diamond is for degree, ellipse is for monomial, rectangle is for equation.

Figure 39: Generate a unique graph representation for a polynomial system




(a) Record the polynomial system on a 4-level graph. Merge variable nodes and degree nodes




(b) Label nodes by numbers

Note: the example is the polynomial system: $x_1 + x_2 + x_3, x_1x_2 + x_2x_3 + x_3x_1, x_1x_2x_3 - 1$.

## 7.1 A graph representation for the polynomial system

A type of graph with labelled vertices is designed to represent polynomial systems. We convert a polynomial system to its unique graph representation in the following steps:

1. Expand and simplify equations of the polynomial system.
2. Generate a tree of 4 levels, i.e. equation, monomial, degree and variable, to record the system by graph nodes. Equation nodes are connected to monomial nodes, monomial nodes are connected to degree nodes, and degree nodes are connected variable nodes. In a monomial, each variable has one degree. Thus, one variable node are connected to one degree node. See Figure 38 (a).
3. Merge the nodes of the same variable and the degree nodes of the same degree for the same variable. See Figure 38 (b).
4. Label nodes by numbers. Label each equation node by its constant, label each monomial node by its coefficient, label each degree node by its degree number. See Figure 38 (c).

Figure 38 shows the graph construction procedure for a polynomial equation, and Figure 39 shows that for a polynomial system. For convenience, we call this graph GraphPoly.

GraphPoly explicitly stores the information for each monomial and equation. The nodes of GraphPoly are labelled by numbers and the types of nodes, i.e. equation, monomial, degree and variable. In this way, each variable, degree, monomial and equation is represented by one node of its type. Also, the constants of equations, the coefficients of monomials and the degrees of variables are represented by the numbers on their nodes. Moreover, the edges have one to one-to-one mapping to the relationships between equations, monomials, degrees and variables.

**Theorem 1.** *Each polynomial system has a unique graph representation, up to isomorphism.*

*Proof.* The proof is straight forward from the graph construction procedure and it goes in two ways.

From the procedure, like Figure 38, there is a bijection between the graph nodes and the elements of the polynomial system, such as equations, monomials, degree and variable. Also, there is a bijection between the graph edges and the relationships between equations, monomials, degrees and variables.

If for two polynomial systems, $F_1$ and $F_2$, we can construct the same graph of polynomial system. For both polynomial systems, there is a bijection from each equation, monomial, degree and variable to the nodes on the graph. Thus, there is a bijection between two polynomial systems for each of their equations, monomials, degrees and variables. So $F_1$ and $F_2$ are exactly the same.

In the other way, if there are two graphs from the same polynomial system. For both graphs, there is a one-to-one mapping from the graph nodes to the equations, monomials, degrees and variables of the polynomial system. On the other hand, there is a one-to-one mapping from the graph edges to the relationships of variables, monomials, degrees and variables. Thus, there is a bijection between two graphs for both nodes and edges. So, two graph are isomorphic.

Note: the bijection between a polynomial system and its graph is not unique. For a cyclic-n polynomial system, variables can be permuted, which implies multiple bijections. $\square$

**Theorem 2.** *Isomorphism of polynomial systems is equivalent to graph isomorphism.*

*Proof.* From theorem 1, the isomorphism of two polynomial systems is equivalent to the isomorphism of two vertex-labelled graphs. That is to say, we already prove that the isomorphism of two polynomial systems belongs to graph isomorphism problems.

On the other hand, we want to prove that any undirected unlabelled, unweighted graph can be represented as a polynomial system consisting of 2-variable monomials with degree 1.

Given the graph with vertices $\{N_i\}$, any edge connecting vertex $N_i$ and vertex $N_j$ is represented as a monomial $x_i x_j$. For each connected maximum subset of the graph, we sum all the monomials from its edges as a polynomial equation. For any isolated vertex $N_k$ without any edge to any other vertices, we can use a monomial of one variable to present it as $x_k$. In this way, we represent the entire graph as a unique polynomial system.

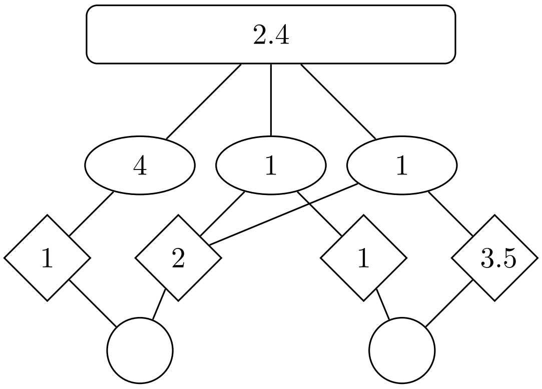
To check whether two undirected non-labelled non-weighted graphs are isomorphic, we can check the isomorphism of these two polynomial systems. □

GraphPoly can also represent the support set of a polynomial systems. We first remove all coefficients of monomials. Then we change the value of an equation node to 1 if a equation has a non-zero constant, otherwise we keep 0. See Figure 40. Corollary 1 and Corollary 2 about the suppport sets directly follow Theorem 1 and Theorem 2. Another proof of Corollary 2 is shown in (8).

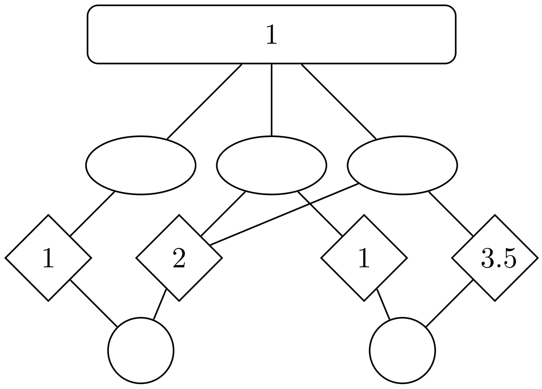
**Corollary 1.** *The support set of each polynomial system has a unique graph representation.*

**Corollary 2.** *Isomorphism of the support set of polynomial systems is equivalent to graph isomorphism.*

Figure 40: A graph representation for a polynomial equation and that of its support set.




(a) A graph representation for a polynomial




(b) A graph representation for its support set

Note: the example is $2.4 + 4x_1 + x_2x_1^2 + x_1^2x_2^{3.5}$ and its support is $1 + x_1 + x_2x_1^2 + x_1^2x_2^{3.5}$.

### 7.1.1 Symmetry of variables on polynomial system graph

For a polynomial system, we care more about the symmetry of variables, which GraphPoly can be used to detect. GraphPoly includes nodes of variable, degree, monomial and degree, but we can simplify GraphPoly to a graph with only nodes of variables. In this way, we can more easily detect the symmetry of variables.

For each two variables, they are related by their common monomials and equations. For GraphPoly, each two nodes of variables are connected by their common nodes of monomial and equation. This is their relationship. After getting all these relationships, we can classify them. Then we store the types of these relationships into the adjacent matrix of the variables. With the adjacent matrix, we can classify it like a graph of only variables and get the symmetry of variables.

The simplified graph of variables can be used to detect the symmetry of variables, but it is not sufficient to prove the symmetry of variables. To prove it, we can use the generators of their symmetric group in DataPoly, as discussed in the following section.

## 7.2 A polynomial data representation by set of set

With set of set, we want to find the unique string representation of a polynomial system.

### 7.2.1 Order of set

If the elements in the set can be sorted, each set has a unique representation. For example,

$$\{1, 2, 3, 0\} \text{ or } \{3, 2, 1, 0\} \xrightarrow{sort} \{0, 1, 2, 3\} \tag{7.2}$$

To compare to two sorted sets $S_1$ and $S_2$, define the order of set by comparing two aspects:

1. Number of elements $len(S)$.

$$len(S1) < len(S2) \Longrightarrow S1 < S2 \tag{7.3}$$

For example,

$$\{4,5,6\} < \{0,1,2,3\}$$

$$\{4,5,6,7,8\} > \{0,1,2,3\}$$

2. The first pair of different elements in two sets. Suppose $S[i]$ is the $i$th element of $S$,

$$\left.\begin{array}{ll} S_1[i] = S_1[i] & \text{if } i < k \\ S_1[k] < S_2[k] & \end{array}\right\} \Longrightarrow S_1 < S2 \tag{7.4}$$

For example,

$$\{0,1,2,3\} < \{0,1,2,4\}$$

$$\{1,3,4,5\} > \{1,2,6,7\}$$

For any two sets are either the same or one is smaller than the other. In this way, all sets have an order to sort. There is a unique sorted representation of any set.

This process can work recursively, i.e. set of set. Each set of set also has a unique representation after sorting. To sort a set of set, we need to sort each element and then sort all of them.

$$\{\{3,2,1\},\{0,5\},\{1,0\}\} \xrightarrow{\text{element sort}} \{\{1,2,3\},\{0,5\},\{0,1\}\} \xrightarrow{\text{sort}} \{\{0,1\},\{0,5\},\{1,2,3\}\} \tag{7.5}$$

To compare two sets of set,

1. Number of elements. For example,

$$\{\{4,5,6,7\},\{7,8,9,10,11\}\} < \{\{0,1,2\},\{0,1,2,3\},\{0,5,6,7\}\} \tag{7.6}$$

2. The first pair of different elements in two sets of set. For example,

$$\{\{0,1\},\{7,8\},\{7,8,9,11\}\} < \{\{0,1\},\{0,1,2\},\{0,1,2\}\} \tag{7.7}$$

In this way, set of set, set of set of set, etc, has a unique representation after sorting.

The exponent structure of a polynomial system has a unique representation. First, we find all types of the monomial exponents. Consider a exponent set of all monomials, after sorting, it has an unique representation as a set of set.

$$\{\prod_j x_*^{a_j^i}, a_j^i \neq 0\}_i \xrightarrow{\text{exponent}} \{\{a_j^i\}_j\}_i \tag{7.8}$$

For example,

$$\begin{aligned} \{x_*^2 x_*^1, x_*^3, x_*^1, x_*^2 x_*^3\} &\xrightarrow{\text{exponent}} \{\{2,1\},\{3\},\{1\},\{2,3\}\} \\ &\xrightarrow{\text{sort}} \{\{1\},\{3\},\{1,2\},\{2,3\}\} \end{aligned} \tag{7.9}$$

Then, we find all types of the equation exponents. For each equation,

$$f(\mathbf{x}) = C + \sum_{\mathbf{a} \in A} c_{\mathbf{a}} \mathbf{x}^{\mathbf{a}}, \quad c_{\mathbf{a}} \neq 0, \tag{7.10}$$

we can identify the type of each monomial in Equation 7.9. For example,

$$x_*^1 x_*^2 + x_*^1 x_*^2 + x_*^1 + C \xrightarrow{\text{exponent}} \{\{1,2\},\{1,2\},\{1\},C\} \tag{7.11}$$

$$\xrightarrow{\text{mon type}} \{C, 1, 3, 3\} \tag{7.12}$$

Finally, we identify all types of equations for the entire polynomial systems. For example,

$$\begin{aligned} f_1 = &\quad x_1 x_2^2 + x_1 x_3^2 \quad + x_1 + 1 \\ f_2 = &\quad x_2 x_1^2 + x_2 x_3^2 \quad + x_2 + 1 \\ f_3 = &\quad x_1^3 \quad + x_3 + 1 \end{aligned} \tag{7.13}$$

1. Record the support of the polynomial system

$$\begin{aligned} f_1 = &\quad x_*^1 x_*^2 + x_*^1 x_*^2 \quad + x_*^1 + C \\ f_2 = &\quad x_*^1 x_*^2 + x_*^1 x_*^2 \quad + x_*^1 + C \\ f_3 = &\quad x_*^3 \quad + x_*^1 + C \end{aligned} \tag{7.14}$$

2. Classify monomials, equations and identify system

$$
\begin{aligned}
MonType &= \{x_*^1, x_*^3, x_*^1 x_*^2\} \\
EqType &= \{\{C, m_1, m_2\}, \{C, m_1, m_3, m_3\}\} \\
Sys &= \{eq_1, eq_2, eq_2\}
\end{aligned}
\tag{7.15}
$$

Because al types of monomial, equation and system has set structures, they can be sorted. Thus, the exponent structure of a polynomial system has a unique representation.

Given an order of variables, we can find a unique representation of a polynomial system. All indices can be recorded as set of set. Elements of the same degree, monomial type or equation type can be switched. After sorting these elements, we have a unique representation of variable indices.

$$
\begin{aligned}
f_1 = &\quad {x_1}^1 {x_2}^2 + {x_1}^1 {x_3}^2 \quad -{x_1}^1 + 1 \\
f_2 = &\quad {x_2}^1 {x_1}^2 + {x_2}^1 {x_3}^2 \quad -{x_2}^1 + 1 \\
f_3 = &\quad {x_1}^3 \quad -{x_3}^1 + 1
\end{aligned}
\tag{7.16}
$$

$$
Index = \{([1],[3]), ([1],[1,2],[1,3]), ([2],[2,1],[2,3])\} \tag{7.17}
$$

By combining Equation 7.15 and 7.2.2, we get the unique representation of a polynomial system. For the equation constant, the first element of $EqType$ is 1, if the equation type has a constant, otherwise it is 0. Here is the data structure DataPoly for Equation 7.14.

$$\begin{aligned} MonType &= \{((1),(3),(1,2)\} \\ EqType &= \{(1,1,2),(1,1,3,3)\} \\ Sys &= \{1,2,2\} \\ Index &= \{([1],[3]),([1],[1,2],[1,3]),([2],[2,1],[2,3])\} \end{aligned} \tag{7.18}$$

It is straight forward that each DataPoly and each polynomial system has 1-to-1 mapping. When the order of monomials and equations are switched, DataPoly keeps the same.

### 7.2.2 Check permutations on DataPoly

For any two permutations $p1$ and $p2$ of variables in a polynomial system, they are the same for a polynomial system, if $\mathbf{f}(p1(\mathbf{x})) = \mathbf{f}(p2(\mathbf{x}))$.

For each permutations, it defines an order of variables. Thus, it has the unique representation of DataPoly, especially for $Index$ in. If two permutations generates the same, they are the same for the polynomial system. It takes $O(n\log(n))$ to check whether two permutations are the same by DataPoly.

### 7.2.3 A unique string representation of a polynomial system

If variables are allowed to switch, we can still find the unique representation by $O(n!)$, where $n$ is number of variables. The number of the variables' orders are $n!$. Each order of variables has a unique $Index$. It is easy to compare these $Index$ and find the minimum as the representation.

To reduce complexisty, we can combine DataPoly and GraphPoly to decrease complexity. Refinement is used to classify different types of nodes by its own type and its neighbors' types. After refinement, we can identify different types of variables. The variables are partitioned by different types. So we can avoid permutations between different partitions. On the other hands, when we try permutations, if we fix a variable node by giving it an index, the other unfixed variables are influenced after the refinement of the GraphPoly. Thus, this also creates more partitions of variables. Through the refinement of GraphPoly, we can largely reduce the complexity of checking all permutations.

## 7.3 A polynomial database and search engine

Traditional databases use text to store the polynomial systems. But based on text, it is hard to search a polynomial system from the database. With the canonical form of GraphPoly, we create tables of polynomial equations, polynomial systems and their relationships. Users can search the database by the following keywords:

1. a polynomial system or its support
2. equations in a polynomial system or their supports
3. monomials' supports

# CITED LITERATURE


1. Adrovic, D.: Solving Polynomial Systems With Tropical Methods. Doctoral dissertation, University of Illinois at Chicago, 2012.

2. Adrovic, D. and Verschelde, J.: Computing Puiseux series for algebraic surfaces. In Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation (ISSAC 2012), eds, J. van der Hoeven and M. van Hoeij, pages 20–27. ACM, 2012.

3. Adrovic, D. and Verschelde, J.: Polyhedral methods for space curves exploiting symmetry applied to the cyclic $n$-roots problem. In Proceedings of CASC 2013, eds, V. Gerdt, W. Koepf, E. Mayr, and E. Vorozhtsov, pages 10–29, 2013.

4. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., and Tomov, S.: QR factorization on a multicore node enhanced with multiple GPU accelerators. In Proceedings of IPDPS 2011, pages 932–943. IEEE Computer Society, 2011.

5. Anderson, M., Ballard, G., Demmel, J., and Keutzer, K.: Communication-avoiding QR decomposition for GPUs. In Proceedings of IPDPS 2011, pages 48–58. IEEE Computer Society, 2011.

6. Backelin, J.: Square multiples n give infinitely many cyclic n-roots. Reports, Matematiska Institutionen 8, Stockholms universitet, 1989.

7. Bates, D. J., Hauenstein, J. D., Sommese, A. J., and Wampler, C. W.: Bertini: Software for numerical algebraic geometry. Available at `http://www.nd.edu/~sommese/bertini/`.

8. Bliss, N., Sommars, J., Verschelde, J., and Yu, X.: Solving polynomial systems in the cloud with polynomial homotopy continuation. In Computer Algebra in Scientific Computing, 17th International Workshop, CASC 2015, Aachen, Germany, September 14-18, 2015, Proceedings, eds, V. Gerdt, W. Koepf, W. Seiler, and E. Vorozhtsov, volume 9301 of Lecture Notes in Computer Science, pages 87–100. Springer-Verlag, 2015.

9. Bouillaguet, C., Fouque, P.-A., and Véber, A.: Graph-theoretic algorithms for the "Isomorphism of Polynomials" problem. In EUROCRYPT 2013, eds, T. Johansson

and P. Nguyen, volume 7881 of Lecture Notes in Computer Science, pages 211–227. Springer-Verlag, 2013.

10. Brandes, T., Arnold, A., Soddemann, T., and Reith, D.: CPU vs. GPU - performance comparison for the Gram-Schmidt algorithm. The European Physical Journal Special Topics, 210(1):73–88, 2012.

11. Chen, T., Lee, T.-L., and Li, T.-Y.: Hom4PS-3: a parallel numerical solver for systems of polynomial equations based on polyhedral homotopy continuation methods. In Mathematical Software – ICMS 2014, 4th International Conference, Seoul, South Korea, August 4-9, 2014, Proceedings, eds, H. Hong and C. Yap, volume 8592 of Lecture Notes in Computer Science, pages 183–190. Springer-Verlag, 2014.

12. Chen, T. and Li, T.-Y.: Solutions to systems of binomial equations. Annales Mathematicae Silesianae, 28:7–34, 2014.

13. Chen, T. and Mehta, D.: Parallel degree computation for solution space of binomial systems with an application to the master space of $\mathcal{N} = 1$ gauge theories. `arXiv:1501.02237v1 [math.AG] 9 Jan 2015`.

14. Collange, S., Daumas, M., and Defour, D.: Interval arithmetic in CUDA. In GPU Computing Gems Jade Edition, ed. W. mei W. Hwu, pages 99–107. Elsevier, 2012.

15. Dai, Y., Kim, S., and Kojima, M.: Computing all nonsingular solutions of cyclic-n polynomial using polyhedral homotopy continuation methods. J. Comput. Appl. Math., 152(1-2):83–97, 2003.

16. Datta, R.: Finding all nash equilibria of a finite game using polynomial algebra. Economic Theory, 42(1):55–96, 2009.

17. Demmel, J., Hida, Y., Li, X., and Riedy, E.: Extra-precise iterative refinement for overdetermined least squares problems. ACM Trans. Math. Softw., 35(4):28:1–28:32, 2009.

18. Faugère, J. C.: Finding all the solutions of Cyclic 9 using Gröbner basis techniques. In Computer Mathematics - Proceedings of the Fifth Asian Symposium (ASCM 2001), volume 9 of Lecture Notes Series on Computing, pages 1–12. World Scientific, 2001.

19. Faugère, J.-C. and Perret, L.: Polynomial equivalence problems: Algorithmic and theoretical aspects. In EUROCRYPT 2006, ed. S. Vaudenay, volume 4004 of Lecture Notes in Computer Science, pages 30–47. Springer-Verlag, 2006.

20. Gao, T., Li, T.-Y., and Li, X.: HOM4PS, 2002. Available at `http://www.csulb.edu/~tgao/RESEARCH/Software.htm`.

21. Gonzalez-Vega, L.: Some examples of problem solving by using the symbolic viewpoint when dealing with polynomial systems of equations. In Computer Algebra in Science and Engineering, pages 102–116. World Scientific, 1995.

22. Grabner, M., Pock, T., Gross, T., and Kainz, B.: Automatic differentiation for GPU-accelerated 2D/3D registration. In Advances in Automatic Differentiation, pages 259–269. Springer-Verlag, 2008.

23. Griewank, A. and Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, second edition, 2008.

24. Gunji, T., Kim, S., Kojima, M., Takeda, A., Fujisawa, K., and Mizutani, T.: PHoM – a polyhedral homotopy continuation method for polynomial systems. Computing, 73(4):55–77, 2004.

25. Gupta, S.: Accelerated computing & deep learning. Available at `http://www.hpcadvisorycouncil.com/events/2015/stanford-workshop/pdf/8_NVIDIA.pdf`, 2015.

26. Haque, S. A., Li, X., Mansouri, F., Maza, M. M., Pan, W., and Xie, N.: Dense arithmetic over finite fields with the CUMODP library. In Mathematical Software – ICMS 2014, eds, H. Hong and C. Yap, volume 8592 of Lecture Notes in Computer Science, pages 725–732. Springer-Verlag, 2014.

27. Harris, M.: Optimizing parallel reduction in CUDA. White paper available at `http://docs.nvidia.com`.

28. Harris, M.: How to access global memory efficiently in cuda c/c++ kernels, 2013. Available at `http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/`.

29. Hida, Y., Li, X. S., and Bailey, D. H.: Algorithms for quad-double precision floating point arithmetic. In 15th IEEE Symposium on Computer Arithmetic (Arith-15

2001), 11-17 June 2001, Vail, CO, USA, pages 155–162. IEEE Computer Society, 2001. Shortened version of Technical Report LBNL-46996, software at `http://crd.lbl.gov/~dhbailey/mpdist`.

30. Huber, B., Sottile, F., and Sturmfels, B.: Numerical Schubert calculus. J. Symbolic Computation, 26(6):767–788, 1998.

31. Huber, B. and Verschelde, J.: Pieri homotopies for problems in enumerative geometry applied to pole placement in linear systems control. SIAM J. Control Optim., 38(4):1265–1287, 2000.

32. Huber, B. and Sturmfels, B.: A polyhedral method for solving sparse polynomial systems. Mathematics of Computation, 64(212):pp. 1541–1555, 1995.

33. Kelley, C.: Solution of the Chandrasekhar $h$-equation by Newton's method. J. Math. Phys., 21(7):1625–1628, 1980.

34. Kim, M., Rosenthal, J., and Wang, X.: Pole placement and matrix extension problems: A common point of view. SIAM J. Control. Optim., 42(6):2078–2093, 2004.

35. Lee, T. L., Li, T.-Y., and Tsai, C. H.: HOM4PS-2.0: a software package for solving polynomial systems by the polyhedral homotopy continuation method. Computing, 83(2-3):109–133, 2008.

36. Leykin, A.: Numerical algebraic geometry. The Journal of Software for Algebra and Geometry: Macaulay2, 3:5–10, 2011.

37. Li, T.-Y.: Numerical solution of polynomial systems by homotopy continuation methods. In Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics, ed. F. Cucker, pages 209–304. North-Holland, 2003.

38. Li, T.-Y., Wang, X., and Wu, M.: Numerical schubert calculus by the pieri homotopy algorithm. SIAM J. Numer. Anal., 20(2):578–600, 2002.

39. Li, X., Demmel, J., Bailey, D., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S., Kapur, A., Martin, M., Thompson, B., Tung, T., and Yoo, D.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw., 28(2):152–205, 2002.

40. Lu, M., He, B., and Luo, Q.: Supporting extended precision on graphics processors. In Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana, eds, A. Ailamaki and P. Boncz, pages 19–26, 2010. Software at `http://code.google.com/p/gpuprec/`.

41. Maza, M. M. and Pan, W.: Solving bivariate polynomial systems on a GPU. ACM Communications in Computer Algebra, 45(2):127–128, 2011.

42. McKelvey, R. and McLennan, A.: The maximal number of regular totally mixed Nash equilibria. Journal of Economic Theory, 72:411–425, 1997.

43. Mukunoki, D. and Takashashi, D.: Implementation and evaluation of triple precision BLAS subroutines on GPUs. In Proceedings of the IPDPS Workshops (PDSEC 2012), pages 1372–1380. IEEE Computer Society, 2012.

44. NVIDIA: Cuda toolkit documentation, 2015. Available at `http://docs.nvidia.com/cuda/index.html`.

45. Patarin, J.: Hidden fields equations (HFE) and isomorphism of polynomials (IP): Two new families of asymmetric algorithms. In Advances in Cryptology - EUROCRYPT'96, ed. U. Maurer, volume 1070 of Lecture Notes in Computer Science, pages 33–48. Springer-Verlag, 1996.

46. Rump, S.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica, 19:287449, 2010.

47. Sabeti, R.: Numerical-symbolic exact irreducible decomposition of cyclic-12. LMS Journal of Computation and Mathematics, 14:155–172, 2011.

48. Sommese, A. J., Verschelde, J., and Wampler, C. W.: Numerical irreducible decomposition using PHCpack. In Algebra, Geometry, and Software Systems, eds, M. Joswig and N. Takayama, pages 109–130. Springer-Verlag, 2003.

49. Sommese, A. J. and Wampler, C. W.: The Numerical Solution of Systems of Polynomials: Arising in Engineering And Science. World Scientific Pub Co Inc, 2005.

50. Sottile, F.: Pieri's formula via explicit rational equivalence. Can. J. Math., 49(6):1281–1298, 1997.

51. Sturmfels, B.: Solving Systems of Polynomial Equations. American Mathematical Society, 2002.

52. Szöllősi, F.: Construction, classification and parametrization of complex Hadamard matrices. Doctoral dissertation, Central European University, Budapest, 2011. `arXiv:1110.5590v1`.

53. Verschelde, J.: Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. ACM Trans. Math. Softw., 25(2):251–276, 1999.

54. Verschelde, J.: Modernizing PHCpack through phcpy. In Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), eds, P. de Buyl and N. Varoquaux, pages 71–76, 2014.

55. Verschelde, J. and Yoffe, G.: Evaluating polynomials in several variables and their derivatives on a GPU computing processor. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2012), pages 1391–1399. IEEE Computer Society, 2012.

56. Verschelde, J. and Yoffe, G.: Orthogonalization on a general purpose graphics processing unit with double double and quad double arithmetic. In Proceedings of the 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops (PDSEC 2013), pages 1373–1380. IEEE Computer Society, 2013.

57. Verschelde, J. and Yu, X.: Polynomial Homotopy Continuation on GPUs. ACM Communications in Computer Algebra, to appear.

58. Verschelde, J. and Yu, X.: GPU acceleration of Newton's method for large systems of polynomial equations in double double and quad double arithmetic. In Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC 2014), pages 161–164. IEEE Computer Society, 2014.

59. Verschelde, J. and Yu, X.: Accelerating polynomial homotopy continuation on a graphics processing unit with double double and quad double arithmetic. In Proceedings of the 2015 International Workshop on Parallel Symbolic Computation (PASCO 2015)., pages 109–118. ACM, 2015.

60. Verschelde, J. and Yu, X.: Tracking many solution paths of a polynomial homotopy on a graphics processing unit. In Proceedings of the 17th IEEE International

Conference on High Performance Computing and Communications (HPCC 2015)., pages 371–376. IEEE Computer Society, 2015.

61. Volkov, V. and Demmel, J.: Benchmarking GPUs to tune dense linear algebra. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008. Article No. 31.

62. Watson, L. T., Billups, S. C., and Morgan, A. P.: Algorithm 652: HOMPACK: a suite of codes for globally convergent homotopy algorithms. ACM Trans. Math. Softw., 13(3):281–310, 1987.

63. Watson, L. T., Sosonkina, M., Melville, R. C., Morgan, A., and Walker, H.: HOMPACK90: A suite of Fortran 90 codes for globally convergent homotopy algorithms. ACM Trans. Math. Softw., 23(4):514–549, 1997.

64. Yoffe, G.: Using parallelism to compensate for extended precision in path tracking for polynomial system solving. Doctoral dissertation, University of Illinois at Chicago, 2012. Available at `http://hdl.handle.net/10027/9116`.

# XIANGCHENG YU

xiangchengyu@outlook.com (312)912-2575

## EDUCATION

| | |
|---|---|
| Ph.D. in Mathematical Computer Science - University of Illinois at Chicago | 2010 - 2015 |
| Thesis: Accelerating polynomial homotopy continuation on GPUs | Advisor: Jan Verschelde |
| B.S. in Mathematical Computer Science - Dalian University of Technology | 2009 - 2009 |

## PROFESSIONAL SKILL

| | |
|---|---|
| **Computer Languages** | C/C++, Python, SQL, CUDA, Matlab, Maple |
| **Technical Background** | Parallel and distributed computing, GPU computing, Numerical analysis, Mathematical modeling |

## EXPERIENCE

**Accelerating polynomial homotopy continuation on GPUs** Jan. 2012 - Aug. 2015
*Univ. of Illinois at Chicago* *Chicago, IL*

· Polynomial evaluation and differentiation by a massively parallel algorithm by CUDA
· Multi-precision QR matrix decomposition by CUDA
· Mastered parallel programming with Pthread, MPI

**Cluster Windows User Profile by Machine Learning** May 2014 - Aug. 2014
*Microsoft Internship* *Seattle, WA*

· Data mining from Windows users' feedback to recover usage profile by SQL
· Cluster different types of users by K-mean, PCA and SVM
· Statistical analysis of usage distribution and error estimation

**Web Server for PHCpack** Aug. 2012 - Aug. 2015
*Univ. of Illinois at Chicago* *Chicago, IL*

· Design intuitive user interface for PHCpack to solve polynomial systems by Python CGI and Apache
· Distribute jobs of polynomial systems to multiple cores and computers by TCP server
· Manage users register and login system by MySQL
· Adapt web interface to be accessible by smartphones and tablets by CSS

**Database of Polynomial Systems** Aug. 2012 - Aug. 2015
*Univ. of Illinois at Chicago* *Chicago, IL*

· Construct a database to store solved polynomial systems from web interface
· Reduce the redundancy of polynomial database by optimizing the structure of polynomial systems
· Solve similar polynomial systems from starting systems in polynomial database by homotopy method

**Undergraduate Study and Mathematical Modeling** Aug. 2005 - May 2009
*Dalian University of Technology* *Dalian, China*

· Developed a GUI platform to visualize and path tracking in Homotopy Methods by Matlab
· Applied the population increasing matrix to discuss Chinas birth policy in the next 50 yrs
· Built a model to forecast the water demand in a city

**PUBLICATION**


2015 J. Verschelde and X. Yu. Polynomial Homotopy Continuation on GPUs. ACM Communications in Computer Algebra, to appear

2015 J. Verschelde and X. Yu. Accelerating polynomial homotopy continuation on a graphics processing unit with double double and quad double arithmetic. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation (PASCO 2015).*, pages 109–118. ACM, 2015

2015 J. Verschelde and X. Yu. Tracking many solution paths of a polynomial homotopy on a graphics processing unit. In *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications (HPCC 2015).*, pages 371–376. IEEE Computer Society, 2015

2015 N. Bliss, J. Sommars, J. Verschelde, and X. Yu. Solving polynomial systems in the cloud with polynomial homotopy continuation. In V.P. Gerdt, W. Koepf, W.M. Seiler, and E.V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing, 17th International Workshop, CASC 2015, Aachen, Germany, September 14-18, 2015, Proceedings*, volume 9301 of *Lecture Notes in Computer Science*, pages 87–100. Springer-Verlag, 2015

2014 J. Verschelde and X. Yu. GPU acceleration of Newton's method for large systems of polynomial equations in double double and quad double arithmetic. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC 2014)*, pages 161–164. IEEE Computer Society, 2014

2009 X. Yu, Z. Yin, and D. Zhang. Numerical pde model for design of insulation layer of energy saving building. *Industrial Construction (China)*, (S1):171–174, 2009