Parallel Path Tracking for Homotopy Continuation using GPU

Chiang-Heng Chien
chiang-heng_chien@brown.edu
School of Engineering, Brown University
Providence, RI, 02912, USA

Hongyi Fan
hongyi_fan@brown.edu
School of Engineering, Brown University
Providence, RI, 02912, USA

Elias Tsigaridas
elias.tsigaridas@inria.fr
INRIA
Paris Cedex 05, France

Ahmad Abdelfattah
ahmad@icl.utk.edu
Innovative Computing Laboratory, University of Tennessee
Knoxville, TN, 37996, USA

Stanimire Tomov
tomov@icl.utk.edu
Innovative Computing Laboratory, University of Tennessee
Knoxville, TN, 37996, USA

Benjamin Kimia
benjamin_kimia@brown.edu
School of Engineering, Brown University
Providence, RI, 02912, USA

ABSTRACT

The Homotopy Continuation (HC) method is known as a prevailing and robust approach for solving numerically complicated polynomial systems with guarantees of a global solution. In recent years we are witnessing tremendous advances in the theoretical and algorithmic foundations of HC. Furthermore, there are very efficient implementations of several variants of HC that solve large polynomial systems that we could not even imagine some years ago. The success of HC has motivated approaching even larger problems or gaining real-time performance. We propose to accelerate the HC computation significantly through a parallel implementation of path tracker in both straight line coefficient HC and parameter HC on a Graphical Processing Unit (GPU). The implementation involves computing independent tracks to convergence simultaneously, as well as a parallel linear system solver and a parallel evaluation of Jacobian matrices and vectors. We evaluate the performance of our implementation using both popular benchmarking polynomial systems as well as polynomial systems of computer vision applications. The experiments demonstrate that our GPU-HC provides as high as 28× and 20× faster than the multi-core Julia in polynomial benchmark problems and polynomial systems for computer vision applications, respectively. Code is made publicly available in https://rb.gy/cvcwgq.

KEYWORDS

Polynomial System, GPU, Homotopy Continuation, Numerical Solves, Mathematical Software

ACM Reference Format:
Chiang-Heng Chien, Hongyi Fan, Elias Tsigaridas, Ahmad Abdelfattah, Stanimire Tomov and Benjamin Kimia. 2022. Parallel Path Tracking for Homotopy Continuation using GPU. In *Proceedings of International Symposium on Symbolic and Algebraic Computation (ISSAC '22).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ISSAC '22, July 4–7, 2022, Lille, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Algorithms and efficient implementations for solving systems of polynomial equations (with finite number of solutions) is one of the most important problems in computational algebra. Even more, polynomial systems appear in the whole spectrum of science and engineering [7, 13, 38]. The pervasiveness of polynomial systems demans effective algorithms, and more importantly efficient implementations that can solve systems with many variables and a large number of solutions possibly in real-time, have a significant impact.

There are various approaches for solving polynomial systems. The *symbolic*, also called exact, relies on elimination of variables and mainly exploits resultant as well as Gröbner basis computations [12, 13, 17, 19, 36]. There are various effective implementations of symbolic methods, either standalone or integrated, in computer algebra systems. They are quite efficient for moderate size problems. However, their use in large problems is not practical. In addition, they perform calculations with rationals and so they are not, in general, amenable to floating point calculations and inexact input.

On the opposite side of the symbolic approach lies the world of *numerical* algorithms [7, 15, 27, 38, 39] which can perform computations with floating point numbers and their main ingredient is a variant of the Newton operator. The most prominent variant is the homotopy continuations approach [7, 38, 43] that is the focus of our study. We should also mention the inclusion-exclusion subdivision-based algorithms [8, 14, 29, 33, 48] that try to mimic the procedure of the binary search algorithm. They rely on oracles that certify (or not) the presence of roots in a domain. Different realizations of the oracle lead to different algorithmic variants and implementations. These algorithms are particularly efficient for a moderate number of variables.

In addition to these two classes of approaches, there are also *hybrid symbolic-numeric* algorithms, *e.g.*, [15, 32, 49], that combine the advantages of both the symbolic and numerical algorithms. The main idea is to perform as many computations as possible with low precision. They gradually increase (actually double) the precision when it is not possible to conclude for certain operations and/or certify their results, *e.g.*, [23, 47]. This is an active area of research, with promising results.

We warn the reader that a review on relevant work on polynomial system solvers is enormous and, regretfully, it is not possible to mention all the relevant references here in.

Software for Homotopy Continuation: We focus on one of the most commonly used and highly successful numerical algorithms in practice to solve system of polynomials with a finite number of solutions, namely, Homotopcy Continuation (HC). Homotopy continuation algorithms rely on the simple idea to solve another, easier system, which we call a *start system*, instead of the polynomial system at hand, which we call the *target system*. The key is to observe that the solution changes continuously as the coefficients of the start system are continuously morphed to that of the target system. Each attempt to morph a solution from a start system to a target system creates a track of a curve in the coefficient space; this curve is a solution of an ordinary differential equation. We follow it, or in other words track it, numerically using techniques from numerical algorithms for solving differential equations. We refer the reader to Section 2 for further technical details. One of the main potential advantages of Homotopy Continuation that is exploited in this paper is that it is easily parallelizable. Evidently, we can follow each track independently and in an implementation we can assign it to a different processor or thread.

Currently, there are several efficient and publicly available implementations of (variants of) homotopy continuation algorithms. For example, PHCpack [43] is a complete software package for solving numerically polynomial systems using HC. The core of the library is written in Ada, but it provides interfaces to various environments. It supports various variants of HC and also provide several functionalities besides solving systems with a finite number of solutions; such as mixed volume computations and root counting, numerical irreducible decomposition, and verification. It also supports parallelism in the multicore setting and there is even partial support for GPU computation for evaluating polynomials [44]. As an another example, Bertini [7] is a general purpose solver based on HC that is implemented in C. It supports total-degree and multi-homogeneous-degree start systems and it also provides several functionalities and multiprecision computations. It was used successfully in various problems coming from robotics, for instance. Another example of HC implementation is HOM4PS [11, 27] which is implemented in C++ and provides interfaces to other programming languages and mathematical software. It exploits parallelism through the multicore architecture or distributed environments. *HomotopyContinuation.jl* [9] is probably the most recent addition in our arsenal of HC implementations. It is a software package developed in Julia which runs efficiently, incorporating various advanced algorithmic and programming techniques such as adaptive step size and robust path tracking, and supports inherently (multicore) parallelism.

Our Approach and Contributions: The implementations of HC algorithms have been very successful in practice. However, some practical applications feature systems of polynomials with a large number of variables and appear in high degree so that they are not practical to solve. Furthermore, real-time constraints in applications such as computer vision [10, 21, 34], require solutions in the order of a few milliseconds to be useful in practice. Thus, an efficient method for solving polynomial systems is of paramount importance. Note that the procedure of solving polynomial systems often appears in an inner loop of an iterative method [22, 35].

A central idea for accelerating HC is to execute multiple tracks in parallel. Allison et al [6] used distinct computers as compute nodes, each of which executed one track independently and in parallel. They also considered a variant where a central computer prepared an "inner part" corresponding to track following, which was executed on many computers in parallel.

A second central idea is to parallelize the "inner part" itself. Verschelde and Yu [44–46] proposed parallel strategies for evaluating polynomials and polynomial derivatives. These two ideas in conjunction, namely, parallel execution of tracks and parallelization of portion of each track, lead to considerable speed-up. Note that [44] only focuses on speeding up the computation of one relatively small component, namely, polynomial evaluation. Thus, the overall speed-up is reduced when considering the entire procedure and not as significant as what we propose here. On the other hand, [45] considers multiple tracks in parallel but it is a hybrid approach and requires GPU-CPU cooperation where data transfer between CPU and GPU needed in each HC iteration easily becomes an overhead for the entire HC process.

Glabe and McCarthy [18] noted that a fixed-step size in the context of conditional branching in [44–46] leads to diversification: one delayed process delays all others. They propose a strategy where a track is assigned to a single thread with a common step size and a common number of steps. This leads to some tracks converging but not all. The remaining unconverged tracks are then re-run with a smaller step size. This approach leads to significant speed-up, but also admittedly to a reduction in tracking accuracy. They conclude that "further research is needed to increase speed using a GPU and maintain accuracy". Indeed, this is one contribution of our paper as described below.

Our approach combines the above central ideas of *(i)* executing HC tracks in parallel, and *(ii)* parallelizing the computations within each track. Observe that beyond the polynomial evaluations proposed in [45], there remains ample opportunities to execute computations in parallel. Specifically, this paper designs and implements methods for all components of the prediction-correction steps of HC, namely, *(i)* evaluating the homotopy Jacobian matrix and vector in parallel, and *(ii)* solving a linear system in parallel. Moreover, our GPU-HC can execute both straight-line HC and parameter HC. In particular, we found that the MAGMA linear solver of CUDA was deficient and proposed a kernel merging solution which has led to at least 3× speed-up over vendor design; this development in the form of a modified open source MAGMA is now available for public use.

The performances of the proposed GPU-HC is evaluated using several polynomial benchmark problems and polynomial systems of computer vision applications. Experiment show significant speed-up over the latest HC solver running on multi-core CPUs, as high as 28× and 20× faster than the multi-core Julia in polynomial benchmark problems and polynomial systems for computer vision applications, respectively. Code is made publicly available in https://rb.gy/cvcwgq. The speed-up is not only an attractive performance gain; it is an *enabling gain* that renders previously impractical-to-solve polynomial systems efficiently solvable. It also

enables real-time implementation in problem applications requiring a real-time solution.

Structure of the Paper: In Section 2, we give an overview of the Homotopy Continuation method, and its variant method used in this paper. In Section 3, we introduced the special architecture of a typical GPU that we have to take into account when we design algorithms. In Section 4, we present the implementation details of our GPU-based HC method. Finally, Section 5 reports the performance of our method compared to other state-of-the-art implementation of homotopy continuation methods.

2 HOMOTOPY CONTINUATION

In this section, we present a brief overview of some of the variants of homotopy continuation algorithms implemented on GPU in this paper; we refer to [5, 7, 38] for a thorough introduction on homotopy continuation techniques. Let $F(\boldsymbol{x}) = 0$ be a polynomial system of n equations in n unknowns we wish to solve. The algorithmic approach of **Straight-line Homotopy** is based on a very simple idea. Let $G(\boldsymbol{x}) = 0$ be another polynomial system for which the solutions are "easily" found or already known. Assume that the number of solutions of the two systems are the same. Then, a homotopy is constructed from $F(\boldsymbol{x})$ and $G(\boldsymbol{x})$ via a linear segment, in the space of polynomial systems, as

$$H_t(\boldsymbol{x}) = \gamma\,(1-t)G(\boldsymbol{x}) + tF(\boldsymbol{x}), \qquad \text{for } t \in (0,1), \tag{1}$$

where $\gamma \in \mathbb{C}$ is a random (or sufficiently generic) number. As t goes from 0 to 1, the numerical Homotopy Continuation algorithm traces paths, or curves, emanating from the roots of $G(\boldsymbol{x}) = 0$ to the roots of $F(\boldsymbol{x}) = 0$, Figure 1. Note that it is guaranteed that the solution set of $H(\boldsymbol{x}, t) = 0$ is a set of smooth curves for all $t \in [0, 1]$ [38]; we can reach all the isolated solutions of $H(\boldsymbol{x}, 1) = F(\boldsymbol{x}) = 0$ by following some path starting from $t = 0$ if the starting and target system have the same number of solutions. Also, for almost all choices of complex constant γ, all solution paths defined by the homotopy are regular, *i.e.*, for all $t \in [0, 1)$, the Jacobian matrix of $H(\boldsymbol{x}, t)$ is regular and no path diverges [30].

The tracing of $\boldsymbol{x}(t)$ from each solution of $H(\boldsymbol{x}, 0)$, *i.e.*, $G(\boldsymbol{x})$, can be achieved iteratively. The path can be approximated locally by the solutions of an ordinary differential equation

$$\frac{\partial H}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{dt} + \frac{\partial H}{\partial t} = 0 \longrightarrow \frac{d\boldsymbol{x}}{dt} = -\left(\frac{\partial H}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial H}{\partial t}. \tag{2}$$

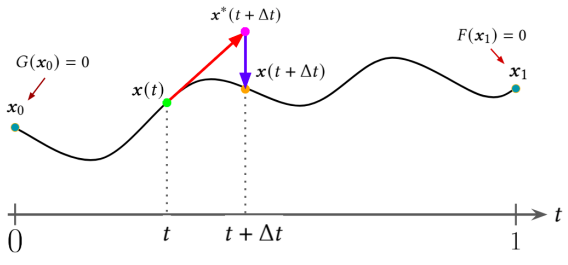

Figure 1: A track (curve) of a Homotopy Continuation algorithm showing $H(\boldsymbol{x}, t)$ in black, along with one prediction (red) and one correction (blue).

This operation is known as **prediction** step in which we follow each path numerically and locally by solving the ODE. Within each iteration, we advance, say by the Euler method, to advance as much as possible and then we use, for example, Newton's Method, to minimize the error of our prediction which is so-called **correction** step. An illustrative concept of prediction and correction steps are shown in Figure 1.

In most of the real-world applications, the polynomial systems depend on a set of parameters, *i.e.*, the coefficients of the polynomial are the function of these parameters. In this case, almost always, the number of solutions is (much) smaller that what Bézout predicts. In addition, we might want to solve the system repeatedly, that is for various specializations of the parameters. Assume that we have a polynomial system $F(\boldsymbol{q}, \boldsymbol{x}) = 0$ that depends on a number of parameters $\boldsymbol{q} = (q_1, \ldots, q_s)$. To exploit homotopy continuation, we first solve the system $F(\boldsymbol{q}_0, \boldsymbol{x}) = 0$ for a random specialization $\boldsymbol{q} = \boldsymbol{q}_0$. Then we have the following homotopy

$$H(\boldsymbol{x}, t) = F(\gamma(1-t)\boldsymbol{q}_0 + t\boldsymbol{q}_1, \boldsymbol{x}) = 0. \tag{3}$$

This homotopy, is known as **Parameter Homotopy** [26, 28, 31]. In this way, starting from the isolated solution of $H(\boldsymbol{x}, \gamma, 0) = F(\boldsymbol{x}, \boldsymbol{q}^0) = 0$ we obtain the isolated solutions of $H(\boldsymbol{x}, \gamma, 1) = F(\boldsymbol{x}, \boldsymbol{q}^1) = 0$.

This technique allows us to pick an arbitrary and not generic $\boldsymbol{q}^0$ as along as $F(\boldsymbol{q}^0, \boldsymbol{x})$ has the same number of solutions as $F(\boldsymbol{q}, \boldsymbol{x})$, for a generic $\boldsymbol{q}$. This allows us to pick an easy starting system. The tracing of the parameter homotopy can also be achieved by the forementioned prediction-correction scheme.

Our Assumptions: In this paper, we mainly focus on the parallelization of the HC path tracking and assume the start systems $G(x)$ and their solutions are known to us on each polynomial system. Moreover, in this work we do not consider about endgames: all our paths track from $t = 0$ to $t = 1$ normally.

3 AN OVERVIEW OF GPU ARCHITECTURE

In this section, we will briefly introduce an overview of a typical GPU architecture and its applications in numerical algorithms in order to give more insights in deploying more numerical algorithms onto GPU. In general, a CPU core is much faster than a GPU core and provides a wider instruction sets. However, GPUs have many more cores; for example, an NVIDIA Tesla V100 GPU has 5,120 cores. Thus, the key to unlock the computational power of the GPU is to design algorithms that are highly parallel and use efficiently as many cores as possible.

Figure 2 shows the Volta architecture used in a Tesla V100 GPU. The CUDA GPU cores are organized into Streaming Multiprocessors (SMs). Each SM has a number of processing blocks with each processing block consisting of a number of CUDA cores. The GPU work is organized into *kernels* that have two levels of nested parallelism: a *coarse level* that is data parallel and is spread across the SMs, and a *fine level* within each SM. The parallelism is organized in terms of thread blocks (TBs). A TB is scheduled for execution on one of the SMs and is data parallel with respect to the other TBs. Each TB is composed of multiple threads running in groups of 32 called *warps*. Threads in a TB can share data through a shared memory module. Private variables that have the scope of one thread are usually stored in the register file. Algorithms must be designed

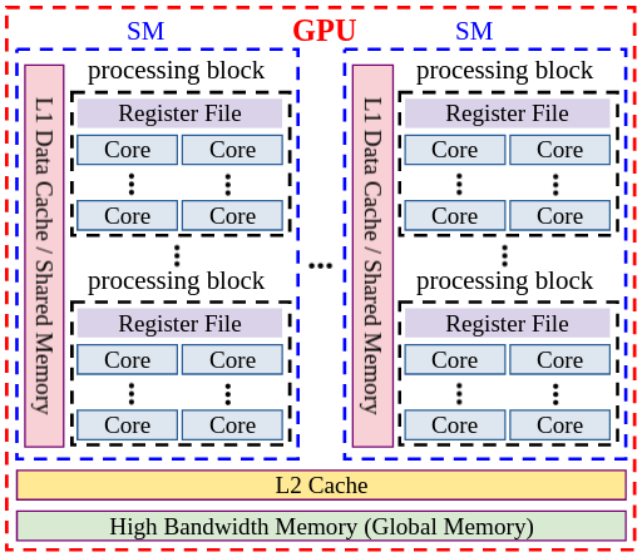

Figure 2: The NVIDIA's GPU architecture and memory hierarchy. In the Tesla V100 GPU using Volta architecture, there are 80 SMs, each partitioned into four processing blocks, each having 16 cores. The register files per SM, shared memory/L1 cache per SM, L2 cache, and global memory are respectively 256KB, 96KB, 6,144KB and 16GB, with hit latencies of 3 *ns*, 22.68 *ns*, 156.33 *ns*, and roughly 366 *ns*, respectively [25].

to support this type of parallelism in order to grasp the parallel computation power from GPUs.

The memory hierarchy of GPUs, *i.e.*, register file, shared memory, L1 and L2 caches, as well as global memory, allows algorithms to reach the compute peak if a certain amount of data is frequently reused. For example, if the data is used only once, the 900 GB/s transfer rate limits the peak performance to 112.5 GFlop/s in double-precision (8 bytes per element). Thus, to reach 7 TFlop/s, an algorithm would need about 64 times data reuse, *i.e.*, each 8 bytes loaded into fast memory must be used in 64 FLOPs. Furthermore, each memory type in GPUs has a distinct data transfer rate, Figure 2. Thus, maximizing data reuse is possible by caching data entirely in the fast memories such as register file or shared memory, which enables GPUs to outperform multicore CPUs.

Numerical algorithms such as homotopy continuation introduced in Section 2, involve many independent computations on relatively small matrices. These algorithms are limited by the memory bandwidth, but have a high degree of parallelism, which is suitable for GPUs. Nevertheless, operations that can not be mapped efficiently to GPU have been left in general for the CPUs [46]. This usually involves irregular computations or computations with significant data dependencies. Still, techniques like batching computations to increase parallelism and developments in numerical linear algebra libraries for GPUs [1, 20], have laid the groundwork for many more algorithms to be easily ported and benefit from the use of GPUs. Algorithms that have been avoided before due to their computational cost can become feasible when current advances make their GPU mapping very efficient. This is the case for HC as described below.

4 GPU IMPLEMENTATION OF HOMOTOPY CONTINUATION

In this work, we present the GPU implementation of two HC methods, **straight-line HC** and **parameter HC**. They share the same prediction-correction algorithmic structure and the tracking of each solution in both methods is independent from other solutions.

4.1 Parallelization of Each Track

A natural parallel intuition is to use one thread per track. However, two issues are required to be considered for efficient GPU processing: *(i)* number of threads used to process many tracks in parallel, and *(ii)* the costly data transfer rates from the register files, L1 caches, L2 cache, and global memory, Figure 2. In a typical homotopy continuation algorithm, each track requires a few Kbytes storing track solutions, start and target coefficients, etc. If we use one thread per track, the available memory in a Volta V100 GPU is 125, 46, 37, and 97K bytes for register file, L1 cache, L2 cache, and global memory, respectively. Therefore, any process requiring memory more than 125 + 46 + 37 = 208 bytes in one track is forward to use global memory which has a very low data transfer rate. As a result, each track must make use of many threads, and not only the processing must be parallelized, but so must the use of memory such that data is kept in memory with fast data transfer rate, *i.e.*, register file, shared memory, or at least L1 cache.

On the other hand, consider an extreme of using numerously many threads per track starts. Although one track access large memory capacity, it is however counterproductive as the synchronization of threads requires the slower shared memory (2 clock cycles per 32 threads) so that if one track occupies 2048 threads, 128 clock cycles (~104 ns) times the tens of thousands of their synchronizations are necessary. This becomes an overhead for the whole HC computation.

To balance the use of the number of threads per track and the time for synchronizing multiple threads, an optimal choice for the HC applications is to assign a track to a warp (32 threads) using one GPU core. Recall that for each SM, at most 64 cores can be employed simultaneously. Thus it gives one track access to $256K/64 = 4K$ very fast register file memory and $96K/64 = 1.5K$ of fast L1 cache (if no shared memory is used), well satisfying the memory requirement of the HC application. On the contrary, the cost of thread synchronization takes only 2 clock cycles.

Given the design of assigning one track to a warp, the algorithm framework is designed as shown in Figure 3. In each prediction step, we aim for solving the following linear system

$$\left(\frac{\partial H}{\partial \boldsymbol{x}}\right)\frac{d\boldsymbol{x}}{dt} + \frac{\partial H}{\partial t} = 0, \tag{4}$$

for getting the local derivative of the path $\boldsymbol{x}(t)$ as

$$\frac{d\boldsymbol{x}}{dt} = -\left(\frac{\partial H}{\partial \boldsymbol{x}}\right)^{-1}\frac{\partial H}{\partial t} = g(\boldsymbol{x}, t). \tag{5}$$

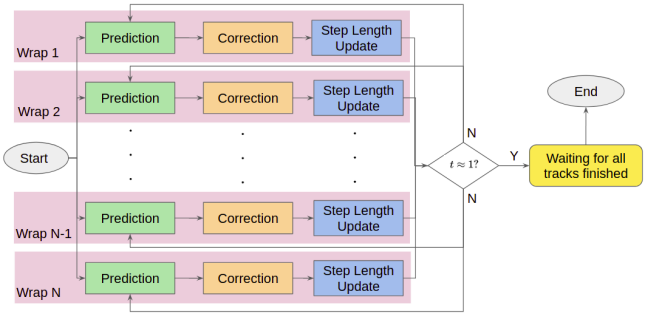

Figure 3: The flow chart of our parallelization scheme.

Then we obtain the prediction by solving the ODE using 4th-order Runge-Kutta method:

$$\boldsymbol{x}^*(t+\Delta t) = \boldsymbol{x}(t) + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right), \tag{6}$$

where

$$\begin{cases} k_1 &= g(\boldsymbol{x}, t) \\ k_2 &= g(\boldsymbol{x} + \frac{k_1}{2}\Delta t, t + \frac{1}{2}\Delta t) \\ k_3 &= g(\boldsymbol{x} + \frac{k_2}{2}\Delta t, t + \frac{1}{2}\Delta t) \\ k_4 &= g(\boldsymbol{x} + k_3\Delta t, t + \Delta t). \end{cases}$$

The correction step is trying to correct the initial prediction onto the solution track $\boldsymbol{x}(t)$. Within each iteration, we use Newton's method to minimize the error between prediction and the path $\boldsymbol{x}(t)$. Each iteration in Newton's method is

$$\hat{\boldsymbol{x}} = \boldsymbol{x}^*(t+\Delta t) - f(\boldsymbol{x}^*, t+\Delta t), \tag{7}$$

where $f(\boldsymbol{x}^*, t)$ is the solution of the following linear system

$$\frac{\partial H(\boldsymbol{x}^*, t+\Delta t)}{\partial \boldsymbol{x}} f(\boldsymbol{x}^*, t+\Delta t) = H(\boldsymbol{x}^*, t+\Delta t). \tag{8}$$

Note that when the step length Δt is too large, the correction step cannot converge in most of the time. In the meantime, when Δt is too small, the whole tracking needs redundant iterations to converge to $t = 1$. In this work, we use a simple but effective adaptive step length control scheme: with an initial Δt, if the correction step converges 20 times, then we double the step length, otherwise, once the correction step fails, the step length is cut half.

4.2 Parallelization of Linear Solver and Jacobian Evaluation

The core computation in the path tracking scheme are *(i)* the solution of the linear system, Equation 5 and 8; and *(ii)* the evaluation of Jacobian $\frac{\partial H}{\partial \boldsymbol{x}}$ and $\frac{\partial H}{\partial t}$. These two operations can further be accelerated by parallelization using a GPU core.

(i) Linear System Solver: The vast majority of work on solving linear systems on GPU is centered around large matrices, normally the size is more than 1000×1000, motivating a hybrid CPU+GPU approach [41, 42]. However, in our HC application, the linear system has a smaller size, normally less than 32×32 (The size is the same with the number of unknowns). In this case, some existing software packages/libraries, such as cuBLAS or MAGMA [2, 3, 24] can be used.

In these packages, a linear system is generally solved by an LU factorization with partial pivoting followed by two triangular solves. The LU factorization of a matrix typically proceeds by summing the first row and others to zero out all but the first row in the first column, then proceed to zero out all but the first two rows in the second column, etc. The LU factorization of a matrix proceeds one column at a time. For each column, *(i)* a *pivot* is chosen based on the maximum absolute value, *(ii)* a row interchange is applied so that the pivot element is brought on the diagonal, *(iii)* the current column is scaled with respect to the pivot, and *(iv)* a rank-1 update is applied to the trailing matrix. After an LU factorization, two triangular solves are in charge of solving the system using L and U factors to achieve forward and backward substitutions.

The LU factorization in MAGMA is fast, typically 15% to 80% faster than cuBLAS for small matrices, namely $12\mu s$ and $38\mu s$ for 4×4 and 20×20 matrices. This is because for matrices less than 32×32, original MAGMA routine assigns one row per thread in a warp, and the factorization proceeds one column at a time in the register file while inter-thread communication occurs in shared memory. Nevertheless, in terms of the combined (factorization + solve) operation, we found out that cuBLAS is faster than MAGMA. This is mainly due to a slow triangular solver kernel in MAGMA, which does not take advantage of small matrices.

Our contribution to improving these standard libraries for our purposes is twofold. First, factorization and solve kernels can be further fused for maximizing the data reuse when the size of the matrices are small, preventing from writing the factorized matrix into the global memory of the GPU and then reading it back in another kernel. The proposed *kernel fusion* therefore significantly speeds up the solution.

Second, in solving a linear system $Ax = b$, the LU decomposition can act on the augmented matrix $[A\ b]$, which implicitly carries out the triangular solve with respect to the L factor of A so that as A is being transferred to the triangular matrix U, so is b, effectively factoring out the L portion which is now done in the decomposition process. The second triangular solve uses the cached U factor after the factorization is complete. This saves the time from the triangular solves step. The proposed fused kernel is now integrated into the MAGMA library as part of its batch routine.

(ii) Parallel Evaluations of the Jacobian Matrix and Vector: The main bottleneck to parallel evaluations of the elements of the Jacobian matrix $\partial H/\partial x$ and the vectors $\partial H/\partial t$ and H is the heterogenuity computations of its elements because it prevents evaluations by many threads requiring a uniform format that enables the GPU to compute in a single instruction multiple threads fashion. This heterogenuity can be illustrated by a simple example of a system with two variables $X = (x_1, x_2)$ where the Jacobian elements are spanned by monomials, for example, in a straight-line HC, $f_1 = c_1x_1 + c_2x_1^2 + c_3x_2^2$ and $f_2 = c_4x_1x_2 + c_5x_1^2$, where the coefficients c_i are linear interpolation of corresponding monomials in the start and target systems. The straightforward approach to homogenize these expressions is to write each as a sum over all possible monomials and associate a scalar zero with those absent from the Jacobian elements in parallel. However, the extreme sparsity from all possible monomials in each element prevents the evaluation process from being computed efficiently.

As an alternative approach, consider K the maximum number of terms in the Jacobian matrix elements; in the above examples, f_1 has three terms and f_2 has two terms, so that $K = 3$ if these were the only elements of the Jacobian matrix. In addition, consider that each term consists of a scalar multiplied with a coefficient and a number of variables, *e.g.*, the third term of f_1 is a product of $(1, c_3, x_2, x_2)$ while the first term of f_2 is $(1, c_4, x_1, x_2)$. Note that the first term of f_1 is a product of $(1, c_1, x_1)$. Thus, to homogenize the expression, it is written as $(1, c_1, x_1, x_3)$ where the auxiliary variable $x_3 = 1$. Likewise, the third term of f_2 is zero, and thus its homogenized expression is $(0, 1, 1, 1)$. As a result, all terms of both f_1 and f_2 can be written as

$$Q = \sum_{k=1}^{K} s_k c_{k,j} x_{k,m_1} x_{k,m_2} \cdots x_{k,m_M}, \tag{9}$$

where s_k is a binary scalar, $c_{k,j}$ represents a coefficient, x_{k,m_i} identifies one of the variables, including $x_3 = 1$, and M is the maximal number of variables in a term. With this in mind the only data to be communicated for the parallel evaluation of Q is $(s_k, c_{k,j}, x_{k,m_1}, x_{k,m_2}, ..., x_{k,m_M})$ where $c_{k,j}$, x_{k,m_i} are pointers to data stored in shared memory and accessed from an array of indices, *i.e.*, f_1 is represented by $((1, 1, 1, 3), (1, 2, 1, 1), (1, 3, 2, 2))$ and f_2 is represented by $((1, 4, 1, 2), (1, 5, 1, 1), (0, 1, 1, 1))$, respectively. Note that $\partial H/\partial t$ and H are evaluated in the same way although the coefficients c_k are different. This homogeneous form allows for parallel computation of all elements of $\partial H/\partial x$, $\partial H/\partial t$, and H.

The idea of parallel evaluations can also be applied to parameter HC, except that the coefficients c_i are functions of start and target parameters. The parallel computation of coefficients c_i can be illustrated by another simple example of a system with the same two variables $X = (x_1, x_2)$. Let $f_1 = c_1x_1 + c_2x_1^2 + c_3x_2^2 = p_1p_2x_1 + p_3x_1^2 + (p_4 + p_5)x_2^2$ and $f_2 = c_4x_1x_2 + c_5x_1^2 = (p_1 + p_3)x_1x_2 + (p_2 - p_4)x_1^2$, where the parameters p_j are linear interpolation of corresponding start and target parameters. To homogenize the computation of coefficients c_i from parameters p_j, the linear interpolation of parameters are expanded and organized such that c_i becomes a polynomial of uni-variable t, *i.e.*, $c_1 = (p_{t1}t + p_{s1}(1 - t))(p_{t2}t + p_{s2}(1 - t)) = (p_{t1}p_{t2} + p_{s1}p_{s2} - p_{s1}p_{t2} - p_{t1}p_{s2})t^2 + (p_{s1}p_{t2} + p_{t1}p_{s2} - 2p_{s1}p_{s2})t + p_{s1}p_{t2}$, where p_{sj} and p_{tj} are start and target parameters, respectively. With this in mind, each c_i in f_1 and f_2 can be computed in a homogeneous fashion by

$$\mathbf{P} \begin{bmatrix} t^2 & t & 1 \end{bmatrix}^T = \begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 \end{bmatrix}^T, \tag{10}$$

where $\mathbf{P}$ is

$$\mathbf{P} = \begin{bmatrix} p_{t1}p_{t2} + p_{s1}p_{s2} & p_{s1}p_{t2} + p_{t2}p_{s2} - 2p_{s1}p_{s2} & p_{s1}p_{s2} \\ -p_{s1}p_{t2} - p_{t1}p_{s2} & & \\ 0 & p_{t3} - p_{s3} & p_{s3} \\ 0 & p_{t4} - p_{s4} + p_{t5} - p_{s5} & p_{s4} + p_{s5} \\ 0 & p_{t1} - p_{s1} + p_{t3} - p_{s3} & p_{s1} + p_{s3} \\ 0 & p_{t2} - p_{s2} + p_{t4} - p_{s4} & p_{s2} - p_{s4} \end{bmatrix}. \tag{11}$$

Since p_{sj} and p_{tj} do not change throughout the entire HC process, the only variable is t in equation (10), and thus, $\mathbf{P}$ is constant and can be computed prior to the parameter HC execution. As a result, before parallel evaluations of the Jacobian matrix and the vector, each coefficient can be computed simultaneously.

Finally, there is an issue on how to allocate the parallel evaluations of the Jacobian matrix $\partial H/\partial x$ and the vectors $\partial H/\partial t$ and H per thread. Recall that each HC track is assigned to a warp which has 32 threads. Since the matrices are generally less than 32×32, and since the subsequent operation of LU decomposition is row-by-row with one thread per row, it makes sense to assign one row per thread.

5 EXPERIMENTAL RESULTS

Four experiments are considered in this paper including *(i)* parallel batch linear systems using kernel fusion, *(ii)* performances and comparisons of the proposed GPU-HC on selected benchmark polynomial systems using coefficient HC, *(iii)* performances and comparisons of the proposed GPU-HC on the katsura polynomial systems, and *(iv)* performances and comparisons of the proposed GPU-HC on polynomial systems for computer vision applications using parameter HC. We use an 8-core 2.6GHz Intel Xeon CPU for all CPU computed solvers and an nVidia Quadro RTX 6000 GPU for our GPU-HC, unless otherwise specified.

All the performances of our GPU-HC were compared with the state-of-the-art HC solvers running on 8 cores CPU, including Julia's *HomotopyContinuation.jl* package [40] version 2.6.3, PHCpack [43] matlab wrapper version 2.4.72, and Minimial problem Numerical continuation Solver (MiNuS) [16] version 1.0. Because the katsura problems has number of solutions equivalent to its total degree, Bertini [7] which follows only the number of total degree tracks was employed for comparisons on the katsura problems. To run Bertini using multiple CPU cores, version 1.6 was used due to its availability of using MPI processing. Since our GPU-HC accelerates the tracking part of the HC algorithm, only the time used in the tracking parts of all solvers were compared. In particular, Julia's *BenchmarkTools* package was used to measure Julia solver time, a matlab timer was employed to measure PHCpack solver time, and C++ high resolution timer was used for MiNuS and our GPU-HC. The time measured in each polynomial problem is an average of 100 runs. In addition, the settings of each solver are by default, except that the tolerance of numerical accuracy of converged tracks for all solvers is set as 10^{-8}.

5.1 Parallel Linear System Solver

The performances of the parallel batched linear systems using the proposed kernel fusion and augmented matrix, Section 4, is compared with cuBLAS and MAGMA with separated kernels as shown in Figure 4. The experiments are conducted on a Tesla V100-PCIe GPU running over 1,000 matrices simultaneously with sizes ranging from 4×4 to 20×20. Evidently, the proposed kernel-fused MAGMA outperforms cuBLAS with speedup of 2.23× to 3.65× and MAGMA with separated kernels with speedups ranging from 3.11× to 4.91×.

5.2 Evaluations on Selected Benchmark Polynomial Systems

The performance evaluations of the proposed GPU accelerated coefficient HC and its comparisons with the latest solvers are shown in Table 1, where nine representative polynomial benchmark systems were selected for comparisons. The start system of each polynomial problem was created via Julia with random coefficient inputs, and for each problem, the start system is the same for all the solvers. A review of Table 1 which is ordered by the number of unknowns,

Table 1: Performances of Julia, PHCPACK, MiNuS, and the proposed coefficient GPU-HC on selected benchmark problems.

problems	# of unkns.	# of sols.	$\partial H/\partial x$		$\partial H/\partial t$		GPU-HC max steps	time (ms)				speedup $\left(\frac{\text{Julia}}{\text{GPU-HC}}\right)$
			K	M	K	M		Julia	PHCpack	MiNuS	GPU-HC	
alea6	6	387	4	2	6	3	31	25.26	179.64	105.67	2.02	×12.50
game6two	6	265	16	4	32	5	77	68.33	660.87	93.85	13.18	×5.49
game7two	7	1854	32	5	64	6	68	1026	9510.68	537.15	153.15	×10.52
cyclic7	7	924	6	6	7	7	24	17.27	358.96	177.95	4.77	×4.58
cyclic8	8	1152	7	7	8	8	24	27.24	1029.53	227.03	7.17	×6.00
cyclic9	9	5994	8	8	9	9	24	166.74	7867.78	1066.7	74.43	×3.81
d1	12	48	3	2	7	3	29	13.61	131.06	61.72	2.48	×5.18
eco12	12	1024	11	2	12	3	24	132.39	2182.15	227.54	17.06	×13.76
pole28sys	16	12862	8	1	73	2	278	50102	439245.99	71387	5354.7	×9.36

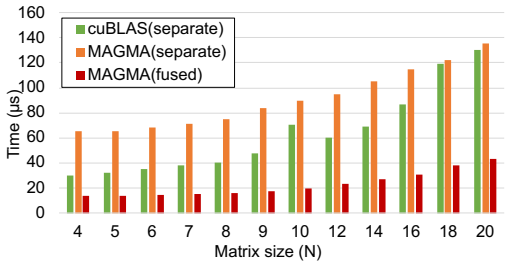

Figure 4: The performances of parallel linear systems using MAGMA with kernel fusion, MAGMA with separated kernels, and cuBLAS.

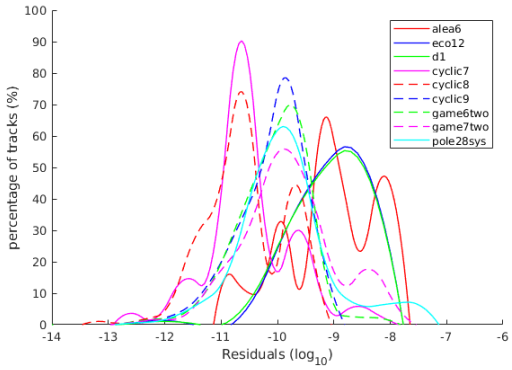

Figure 5: Numerical accuracy distribution of polynomial benchmark systems.

shows that GPU-HC outperforms all the other solvers. Specifically, when compared to Julia, GPU-HC provides speedup ranging from around 3.8× to 13.9×.

Numerous factors affect the speedup of GPU-HC over Julia, including the number of solutions, the maximal steps among all tracks in GPU-HC, and the number of terms as well as variables in each term in the evaluations of $\partial H/\partial x$ and $\partial H/\partial t$, *i.e.*, K and M, equation (9). Basically, if the number of solutions is larger, the parallelism of following tracks to convergence is more significant. For example, d1 v.s. alea6. Meanwhile, the maximal number of steps required among all tracks in our GPU-HC implies that the GPU execution has to wait for the slowest track before the overall execution is finished. Therefore, the smaller the maximal steps, the better for the GPU-HC. For example, alea6 v.s. game6two. In addition, larger M indicates that when evaluating Jacobian matrix and vector, every thread has to access the variables from shared memory more times than those with small M. In particular, if two threads access the same variable at the same time, an access conflict happens, so these two threads has to take turns; thus the evaluation time takes longer. As a result, faster speed can be obtained for problems with smaller M; for example, eco12 v.s. cyclic8.

Figure 5 shows the numerical accuracy distribution of the benchmark polynomial systems with roots found by GPU-HC. Basically, all the solutions are below 10^{-7}, showing that the speedup is not at the cost of lower accuracy, *i.e.*, the GPU-HC computed solutions satisfy the polynomial system with high accuracy.

5.3 Evaluations on Katusra Polynomial Systems

To show the relationship between the degree of a polynomial system and the speedup made by the proposed GPU-HC, Table 2 shows the katsura family benchmark problem including katsura6 to katsura15 as well as katsura20 and 21. The start systems are also generated by Julia with random coefficients, and are identical for all the solvers. It is clearly shown in Table 2 that as the number of solutions increases, the speedup also increases due to the GPU parallelism of independent tracks executed to convergence. Evidently, our GPU-HC gives 6.9× to 28.33× speedup over Julia for the katsura problems.

Figure 6 shows the graph of the time used by Julia and GPU-HC for katsura6 to katsura15. Basically, Julia's time is exponentially proportional to the polynomial degree, while GPU-HC time is roughly flat from katsura6 to katsura11, and then after katsura11 the time goes exponentially proportional to the degree. This is because in Volta V100, the maximal number of warps that can be run simultaneously is 5120. Thus for problems having more than 5120 solutions, GPU-HC has to compute 5120 solutions in parallel first before the rest of the solutions can be run. However, when the size of register file per SM, *i.e.*, 256 KB, is not affordable for all 64 warps where each warp could occupy at most 256 / 64 = 4 KB, one SM cannot process 64 warps at the same time. For example, in katsura11, each warp needs at least 12 unknowns×(12 unknowns×8B (one row of Jacobian matrix) + 8B (one row of Jacobian vector) + 108×4B (index

Table 2: Performances of Julia, Bertini, PHCPACK, MiNuS, and the proposed coefficient GPU-HC on katsura problems.

problems	# of unkns.	# of sols.	$\partial H/\partial x$		$\partial H/\partial t$		GPU-HC max steps	time (ms)					speedup $\left(\frac{\text{Julia}}{\text{GPU-HC}}\right)$
			K	M	K	M		Julia	Bertini*	PHCpack	MiNuS	GPU-HC	
katsura6	7	64	3	1	8	2	24	4.56	15.82	64.58	64.88	0.66	×6.90
katsura7	8	128	3	1	9	2	24	7.67	39.29	93.86	72.23	0.86	×8.92
katsura8	9	256	3	1	10	2	24	14.66	94.98	140.93	86.93	1.06	×13.83
katsura9	10	512	3	1	11	2	24	25.89	194.88	259.14	117.38	1.76	×14.71
katsura10	11	1024	3	1	12	2	24	55.23	457.34	602.29	201.16	3.82	×14.46
katsura11	12	2048	3	1	13	2	32	120.22	1040.24	1527.54	414.12	7.34	×16.35
katsura12	13	4096	3	1	14	2	32	267.32	2612.97	3401.12	816.76	15.06	×17.75
katsura13	14	8192	3	1	15	2	46	635.49	5071.40	8715.36	1617.43	39.04	×16.28
katsura14	15	16284	3	1	16	2	64	1451	12120.83	18025.23	3330.12	87.02	×16.67
katsura15	16	32768	3	1	17	2	61	3344	28932.94	43232.96	6455.74	169.14	×19.77
katsura20	21	1048576	3	1	22	2	247	247837	1810189.57	2697182.41	276131.9	8457.52	×29.30
katsura21	22	2097152	3	1	23	2	217	497211	4163436.01	6202435.75	557492.90	17550.16	×28.33

*: The time used by Bertini contains the time it takes to write data into files.

array size per row for evaluating the Jacobian matrix) + 52×4B (index array size per row for evaluating the Jacobian vector)) + intermediate variables > 8.9KB which is greater than 4KB. As a result, not all 2048 tracks can be computed concurrently for the katsura11 problem, making the GPU-HC time start to increase exponentially.

The numerical accuracy distribution of the katsura problems solved by GPU-HC can be found in Figure 7. Similar to the benchmark problems, all the solutions are below 10^{-7}, well showing that GPU-HC obtains both satisfying efficiency and accuracy.

5.4 Evaluations on Polynomial Systems for Computer Vision Applications

We select a set of computer vision problems to evaluate the performances of the accelerated parameter HC on GPU, and compare it with the tracking part of the parameter HC in Julia and MiNuS. Table 3 shows the problems covering multi-view triangulation and camera pose estimation. The parameters of the start systems are randomly generated, while the target parameters are generated from the multi-view dataset [16], except for the 6-point rolling shutter absolute pose estimation where its target parameters are given by the source code of [4].

In Table 3, the row and column sizes of $\mathbf{P}$ indicate the number of coefficients and the degree of t, equation (10), respectively. Thus if the degree of t is large, our GPU parameter HC can benefit from the parallelism of computing coefficients from parameters even if the number of solutions is small, and thus give significant speedup over Julia. For example, the 6-point rolling shutter absolute pose estimation problem. Another factor that affects the speedup is the maximal steps. For example, compared with refractive P6P, relaxed 3-view triangulation has similar number of solutions with smaller K and M, but the maximal steps is the double of what refractive P6P has. Thus, GPU-HC requires more time in order to wait for the slowest track to converge. Nevertheless, it is clear from Table 3 that GPU-HC is able to provide speedup ranging from 1.51× to 21.44×.

Figure 8 shows the numerical accuracy distribution of the computer vision problems. Not only all the solutions are below 10^{-7}, it is more importantly that one true solution can be found from all converged tracks. Computer vision problem often requires solving a polynomial system in an iterative loop; for example, a pose estimation problem like refractive P5P, GPU-HC is able to estimate a pose within 1 second using 500 loops. Therefore, GPU-HC enables the HC algorithm to be used in practice.

6 CONCLUSION

This paper designs, develops, and implements GPU-HC, an efficient HC solver implemented on a GPU that accurately solves a system of polynomial equations. The parallelism of the HC algorithm is achieved at two levels: *(i)* a coarse level that follows all the tracks to convergence simultaneously, and *(ii)* a fine level that solves a linear system and also evaluates the Jacobian matrix and related vector within each track in parallel. Both straight-line HC and parameter HC are implemented, where the performances of the straight-line HC is evaluated through a set of benchmark polynomial systems, while the parameter HC is evaluated using polynomial systems arising in computer vision problems. These experiments show a remarkable speedup in both cases. It demonstrates that it is possible to have real-time applications which requires solving a polynomial system in an inner loop of an iterative method such as RANSAC in computer vision. In this sense, GPU-HC is an enabling developments rending problems that were impractical to solve practically.

Acknowledgement

Kimia, Fan, and Chien gratefully acknowledge the support of NSF award 1910530. Tsigaridas is partially supported by ANR JCJC GALOP (ANR-17-CE40-0009). Abdelfattah and Tomov are partially supported by NSF Grant No. OAC 1740250.

REFERENCES

[1] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 21–38, 2016.

Table 3: Performances of Julia, MiNuS, and the proposed parameter GPU-HC on computer vision problems.

problems	# of unkns.	# of sols	size of **P**	$\partial H/\partial x$		$\partial H/\partial t$		GPU-HC max steps	time (ms)			speedup $\left(\frac{\text{Julia}}{\text{GPU-HC}}\right)$
				K	M	K	M		Julia	MiNuS	GPU-HC	
optimal PnP w. quaternion [34]	4	80	107×2	20	3	35	4	53	59.43	55.49	3.47	×17.13
refractive P5P [21]	5	16	90×3	10	2	26	3	43	12.18	63.11	1.75	×6.96
refractive P6P [21]	6	36	108×3	10	3	26	4	62	18.54	67.25	3.75	×4.94
6pt rolling shutter abs. pos. [4]	6	20	96×4	4	1	16	2	40	37.30	54.37	1.74	×21.44
3pt rel. pos. w. homo. constraint [37]	8	8	44×3	6	2	9	3	42	9.99	51.90	2.42	×4.13
relaxed 3-view triangulation [10]	8	31	25×2	3	1	9	2	124	8.74	64.23	5.78	×1.51
3-view triangulation [10]	9	94	34×2	3	1	9	2	158	26.69	150.72	9.75	×2.74
4-view triangulation	11	142	36×2	3	1	9	2	141	48.468	150.20	11.94	×4.06
trifocal rel. pos. w. unknown f	18	1784	153×3	12	4	14	5	130	3964	3326.07	388.27	×10.21

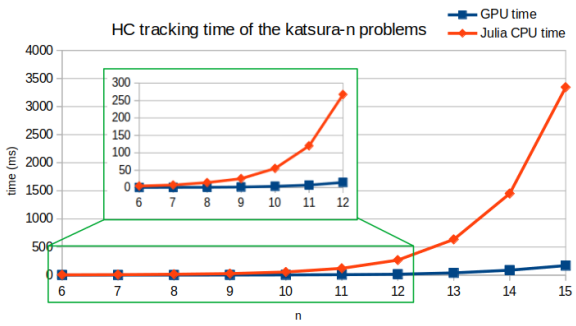

Figure 6: Time used by the proposed GPU-HC and Julia on the katsura-*n* problems.

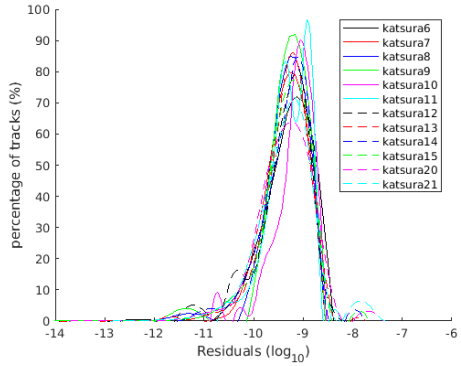

Figure 7: Numerical accuracy distribution of katsura polynomial systems.

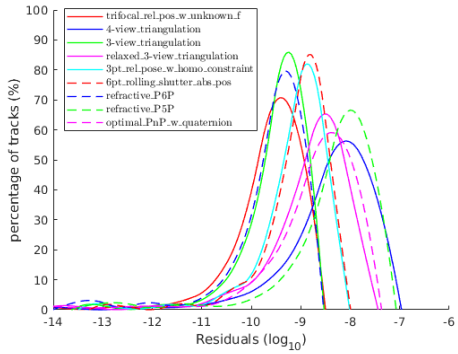

Figure 8: Numerical accuracy distribution of polynomial systems for computer vision applications.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[3] A. Ahmad, H. Azzam, T. Stanimire, and D. Jack. Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures. *Journal of Computational Science*, 26:226–236, 2018.

[4] C. Albl, Z. Kukelova, and T. Pajdla. R6P-rolling shutter absolute camera pose. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2292–2300, 2015.

[5] E. L. Allgower and K. Georg. *Introduction to numerical continuation methods.* SIAM, 2003.

[6] A. C. Allison, Donald CS and L. T. Watson. Granularity issues for solving polynomial systems via globally convergent algorithms on a hypercube. *Journal of Supercomputing*, 3(1):5–20, 1989.

[7] D. J. Bates, A. J. Sommese, J. D. Hauenstein, and C. W. Wampler. *Numerically solving polynomial systems with Bertini.* SIAM, 2013.

[8] R. Becker and M. Sagraloff. Counting solutions of a polynomial system locally and exactly. *arXiv preprint arXiv:1712.05487*, 2017.

[9] P. Breiding and S. Timme. HomotopyContinuation.jl: A package for homotopy continuation in Julia. In J. H. Davenport, M. Kauers, G. Labahn, and J. Urban, editors, *Mathematical Software – ICMS 2018*, pages 458–465, Cham, 2018. Springer International Publishing.

[10] M. Byröd, K. Josephson, and K. Åström. Fast optimal three view triangulation. In *Asian conference on computer vision*, pages 549–559. Springer, 2007.

[11] T. Chen, T.-L. Lee, and T.-Y. Li. Hom4PS-3: a parallel numerical solver for systems of polynomial equations based on polyhedral homotopy continuation methods. In *International Congress on Mathematical Software*, pages 183–190. Springer, 2014.

[12] D. Cox, J. Little, and D. OShea. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra.* Springer Science & Business Media, 2013.

[13] D. A. Cox, J. B. Little, and D. O'Shea. *Using algebraic geometry.* Number 185 in Graduate texts in mathematics. Springer, New York, 2nd edition, 2005.

[14] J.-P. Dedieu and J.-C. Yakoubsohn. Computing the real roots of a polynomial by the exclusion algorithm. *Numerical Algorithms*, 4(1):1–24, 1993.

[15] M. Elkadi and B. Mourrain. Symbolic-numeric methods for solving polynomial equations and applications. In *Solving Polynomial Equations*, pages 125–168. Springer, 2005.

[16] R. Fabbri, T. Duff, H. Fan, M. Regan, D. d. C. de Pinho, E. Tsigaridas, C. Wampler, J. Hauenstein, B. Kimia, A. Leykin, et al. Trifocal relative pose from lines at points and its efficient solution. *arXiv preprint arXiv:1903.09755*, 2019.

[17] M. Giusti, G. Lecerf, and B. Salvy. A Gröbner free alternative for polynomial system solving. *Journal of complexity*, 17(1):154–211, 2001.

[18] J. Glabe and J. M. McCarthy. Numerical continuation on a graphical processing unit for kinematic synthesis. *Journal of Computing and Information Science in*

Engineering, 20(6), 2020.

[19] L. Gonzalez-Vega, F. Rouillier, and M.-F. Roy. Symbolic recipes for polynomial system solving. In *Some tapas of computer algebra*, pages 34–65. Springer, 1999.

[20] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[21] S. Haner and K. Astrom. Absolute pose for cameras under flat refractive interfaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1428–1436, 2015.

[22] A. Hast, J. Nysjö, and A. Marchetti. Optimal RANSAC - towards a repeatable algorithm for finding the optimal set. *Journal of WSCG*, 21(1):21–30, 2013.

[23] J. D. Hauenstein and F. Sottile. Algorithm 921: alphaCertified: certifying solutions to polynomial systems. *ACM Transactions on Mathematical Software (TOMS)*, 38(4):1–20, 2012.

[24] MAGMA: Matrix Algebra on GPU and Multicore Architectures. Available at http://icl.cs.utk.edu/magma/.

[25] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[26] T. Li and X. Wang. Nonlinear homotopies for solving deficient polynomial systems with parameters. *SIAM journal on numerical analysis*, 29(4):1104–1118, 1992.

[27] T.-Y. Li. Numerical solution of multivariate polynomial systems by homotopy continuation methods. *Acta numerica*, 6:399–436, 1997.

[28] T.-Y. Li, T. Sauer, and J. A. Yorke. The cheater's homotopy: an efficient procedure for solving systems of polynomial equations. *SIAM Journal on Numerical Analysis*, 26(5):1241–1251, 1989.

[29] A. Mantzaflaris, B. Mourrain, and E. Tsigaridas. On continued fraction expansion of real roots of polynomial systems, complexity and condition numbers. *Theoretical Computer Science*, 412(22):2312–2330, 2011.

[30] A. Morgan and A. Sommese. A homotopy for solving general polynomial systems that respects m-homogeneous structures. *Applied Mathematics and Computation*, 24(2):101–113, 1987.

[31] A. P. Morgan and A. J. Sommese. Coefficient-parameter polynomial continuation. *Applied Mathematics and Computation*, 29(2):123–160, 1989.

[32] B. Mourrain. Pythagore's dilemma, symbolic-numeric computation, and the border basis method. In *Symbolic-Numeric Computation*, pages 223–243. Springer, 2007.

[33] B. Mourrain, M. N. Vrahatis, and J.-C. Yakoubsohn. On the complexity of isolating real roots and computing with certainty the topological degree. *journal of complexity*, 18(2):612–640, 2002.

[34] G. Nakano. Globally optimal DLS method for PnP problem with Cayley parameterization. In *BMVC*, pages 78–1, 2015.

[35] R. Raguram, J.-M. Frahm, and M. Pollefeys. A comparative analysis of RANSAC techniques leading to adaptive real-time random sample consensus. In *European conference on computer vision (ECCV)*, pages 500–513. Springer, 2008.

[36] F. Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Applicable Algebra in Engineering, Communication and Computing*, 9(5):433–461, 1999.

[37] O. Saurer, P. Vasseur, C. Demonceaux, and F. Fraundorfer. A homography formulation to the 3pt plus a common direction relative pose problem. In *Asian Conference on Computer Vision*, pages 288–301. Springer, 2014.

[38] A. J. Sommese and C. W. Wampler. *The Numerical solution of systems of polynomials arising in engineering and science*. World Scientific, 2005.

[39] H. J. Stetter. *Numerical polynomial algebra*. SIAM, 2004.

[40] S. Timme. Mixed precision path tracking for polynomial homotopy continuation. *Advances in Computational Mathematics*, 47:1–23, 2021.

[41] S. Tomov and J. J. Dongarra. Dense linear algebra for hybrid GPU-based systems. In *Scientific Computing with Multicore and Accelerators*, 2010.

[42] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[43] J. Verschelde. Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. *ACM Transactions on Mathematical Software (TOMS)*, 25(2):251–276, 1999.

[44] 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*, pages 109–118, 2015.

[45] J. Verschelde and X. Yu. Tracking many solution paths of a polynomial homotopy on a graphics processing unit in double double and quad double arithmetic. In *IEEE 17th International Conference on High Performance Computing and Communications, IEEE 7th International Symposium on Cyberspace Safety and Security, IEEE 12th International Conference on Embedded Software and Systems*, pages 371–376, 2015.

[46] J. Verschelde and X. Yu. Polynomial homotopy continuation on GPUs. *ACM Communications in Computer Algebra*, 49(4):130–133, 2016.

[47] J. Xu, M. Burr, and C. Yap. An approach for certifying homotopy continuation paths: Univariate case. In *43rd ACM International Symposium on Symbolic and Algebraic Computation, ISSAC 2018*, pages 399–406. ACM, 2018.

[48] J. Xu and C. Yap. Effective subdivision algorithm for isolating zeros of real systems of equations, with complexity analysis. In *Proc of International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 355–362, 2019.

[49] L. Zhi and G. Reid. Solving nonlinear polynomial systems via symbolic-numeric elimination method. In *In Proceedings of the International Conference on Polynomial System Solving*, pages 50–53, 2004.