A CUDA Library for Modular Polynomial Computation

Sardar Anisul Haque[1] Xin Li[2] Farnam Mansouri[3]

Marc Moreno Maza[4,5] Davood Mohajerani[4] Wei Pan[6]

[1]Qassim University, Saudi Arabia

[2]Universidad Carlos III, Spain

[3]Microsoft, Redmond, USA

[4]ORCCA, University of Western Ontario, Canada

[4]IBM Center for Advanced Studies, Markham, Canada

[6]Intel Corporation, Santa Clara, USA

ISSAC 2017, University of Kaiserslautern, Kaiserslautern, Germany

July 26, 2017

Outline

Outline

Why such a library is even needed?

Accelerating computer algebra algorithms

- Apparently, we like to make our modular computation faster!
- Specially, for dense matrices and polynomials with integer coefficients.
- In 40 years, parallelism has grown into many shapes and forms.
- Supported by multi-core CPUs, many-cores GPUs, accelerators, FPGAs.

General purpose GPU computing using CUDA

- NVIDIA GPUs can be programmed using CUDA.
- CUDA is a powerful platform for writing portable GPU code.
- For the moment, writing efficient parallel programs comes with many subtleties.
- As a rule of thumb, usually there is a trade-off between generality and optimization.
- However, a CUDA library SHOULD be easy to install, learn, and use!

Installing CUMODP

A bit of history

- Licensed under GPL version 3.0
- Version 2.0 (with major re-factoring).

Requirements:

- CUDA 7.0 or later.
- Devices of compute capability 2.0 or above.
- Maple for verification tests and NTL fot benchmarks.
- FLINT is also used for benchmarking, but not required.

How long does it take to be installed?

```
$ wget cumodp.org/latest.tar.gz
$ tar -xzf latest.tar.gz
$ cd cumodp-2.0
$ make install
```

Outline

Outline

Mathematical point of view

Traditionally

- The main focus is on efficiency-critical routines for polynomial system solvers.
- The routines are selected by the observation that polynomial and matrix multiplication are at the core of many algorithms in symbolic computation.
- Expressing the algebraic complexity of an operation in terms of a multiplication time is common.
- "reducing everything to multiplication" [a] is also widely used at the software level, e.g. MAGMA, FLINT, NTL.

[a] Quoting a talk title by Allan Steel

GPUs are more complicated!

- On GPUs, this reduction to multiplication becomes more complex.
- Three complexity measures need to be considered: algebraic complexity (= work), parallelism ($\iff$ span) and parallelism overhead ($\simeq$ cache complexity).

Role of plain algorithms

Plain algorithms on hardware accelerators

- Plain algorithms play a more important role than on single-core processors.
- Parallel versions of plain algorithms often provide similar span (parallel complexity) than their asymptotically fast counterparts!
- In practice, when enough hardware resources (processing cores and memory) are available, parallel plain algorithms can deliver useful performance and outperform their asymptotically fast counterparts.

Design principles of CUMODP

- Mixing plain and asymptotically fast algorithms
- Adaptive implementation: computing w.r.t. available hardware resources.

Three levels of implementation

Level	Exp	Funcionalities	Algorithm Choice
L3	based on L2	advanced arithmetic operations on families of polynomials: operations based on sub-product trees, computations of sub-resultant chains	functions combine several Level 2 algorithms for achieving a given task.
L2	based on L1	basic arithmetic operations for dense or sparse polynomials with coefficients in Z/pZ, Z or in floating point numbers: polynomial multiplication, polynomial division	functions provide several algorithms or implementation for the same operation: coarse-grained & fine-grained, plain & FFT-based.
L1	Plain - asymptotically fast algorithms	basic arithmetic operations that are specific to a polynomial representation or a coefficient ring: multi-dimensional FFTs/TFTs, converting integers from CRA to mixed-radix representations	functions (n-D FFTs/TFTs) are highly optimized in terms of arithmetic count, locality and parallelism.
L0	CUDA Runtime API		
GPU			

Features

Available functions in version 2.0

- A bivariate system solver (integrated into MAPLE),
- Small prime field FFT (1-D & n-D) based on Cooley-Tukey and Stockham algorithms
- Big prime field FFT (1-D) for primes of size up to 16 machine-words
- Univariate polynomial arithmetic (multiplication, division, GCD)
- Sub-product tree, fast multi-point evaluation and interpolation.

In development for next release

- A quick start guide + tutorials
- Big integer arithmetic
- Integer linear programming (ILP)
- Dense linear algebra over finite fields and floating point numbers
- Real root isolation
- More univariate polynomial arithmetic

Outline

Outline

FFT-based multiplication

Stockham FFT algorithm

- Let d be the degree, then $n = 2^{\lceil \log_2(2d-1) \rceil}$.
- Based on the *many-core machine model*, the estimated running time on $\Theta(n)$ processors is in $O(U \log_2(n))$, where $1/U$ is the throughput between one local memory of a processor and the global memory.

Benchmark

Plain multiplication algorithm

Benchmarks

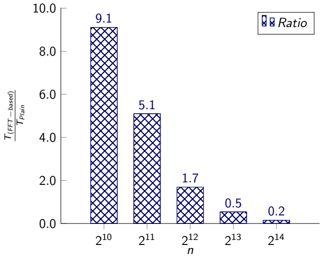


Figure: Ratio (FFT-based/plain) diagram for multiplication of polynomials of degree $n \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$.

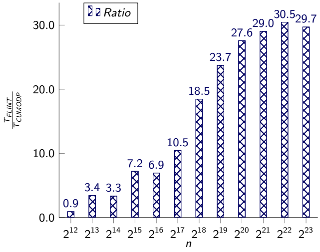


Figure: Ratio (FLINT/CUMODP) diagram for FFT-based multiplication of polynomials of degree $n \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$.

Outline

1. Introduction
2. Design
3. Example: Multiplication
4. Example: Big prime field FFT
5. Example: Sub-product tree
6. Example: Bivariate solver

Outline

Big prime field FFT

Finite fields with large prime characteristics

- Modular methods based on small-prime approaches (using CRT or Hensel lifting) often have to deal with unlucky primes!
- Avoiding unlucky primes is desirable in areas like polynomial system solving.
- This suggests to develop approaches based on larger primes (of size over a machine word) which are more likely to be lucky.

For $\mathbb{Z}/p\mathbb{Z}$ with p as a big prime that fits on k machine-words:

- We use *generalized Fermat primes* of the form $p = r^k + 1$ (where k is a power of 2 and r of machine-word size) to reduce cost of twiddle multiplication.
- Our CUDA code implements arithmetic and FFT over $\mathbb{Z}/p\mathbb{Z}$, for such primes.

Exploiting block parallelism of the Cooley-Tukey FFT algorithm

We expand the six-step recursive FFT algorithm (a variation of Cooley-Tukey):

$$\mathrm{DFT}_N = L_K^N \underbrace{(I_J \otimes \mathrm{DFT}_K)}_{\text{Block parallelism}} L_J^N D_{K,J} \underbrace{(I_K \otimes \mathrm{DFT}_J)}_{\text{Block parallelism}} L_K^N.$$

Benchmarking big prime vs. small prime FFT

Computing FFT: big prime vs. small prime

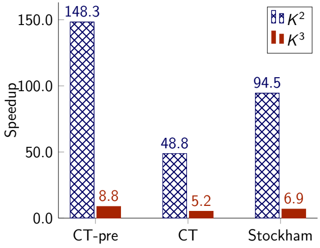
- Small prime approach: pairwise different primes $p_1, \ldots, p_k$
  1. compute image f_i of f in $\mathbb{Z}/p_1\mathbb{Z}[x], \ldots, \mathbb{Z}/p_k\mathbb{Z}[x]$ (*projection*)
  2. compute $\mathrm{DFT}_N(f_i)$ at ω_i in $\mathbb{Z}/p_i\mathbb{Z}[x]$
  3. combine the results using the CRT (*recombination*)
- The small primes are $\frac{\text{machine-word size}}{2}$ $\Rightarrow$ fair to use $2k$ of them!

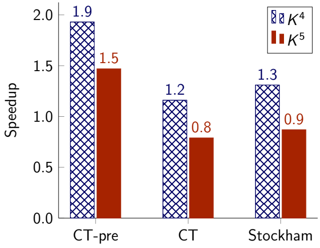
Small prime (32-bit Fourier primes) FFTs from the CUMODP library:

- Computing DFT_N for $N = 2^n$ with $8 \le n \le 26$
- We use the CT, the CT with pre-computed powers of ω, and the Stockham FFT.
- Computing the big prime field FFTs over $\mathbb{Z}/p\mathbb{Z}$, with $P_3 = (2^{63} + 2^{34})^8 + 1$.
- We use 16 and 32 small primes for comparing against P_3 and P_4, respectively.
- Tests completed on a NVIDA GeforeGTX760M card

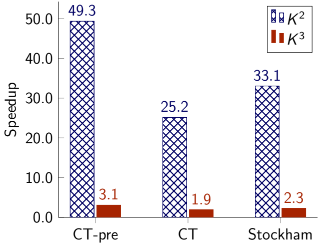
Small primes approach / Big prime approach

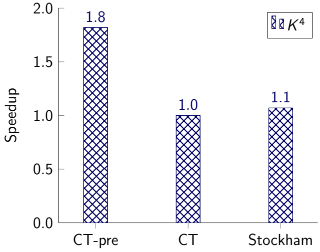
Benchmarking for $P_3 = (2^{63} + 2^{34})^8$ with FFT bases-case $K = 16$





Benchmarking for $P_4 = (2^{62} + 2^{36})^{16}$ with FFT bases-case $K = 32$





Outline

Outline

Subproduct tree

Subproduct tree techniques for multi-point evaluation and interpolation

- Standard implementations of subproduct tree techniques are pure serial code, including the NTL (for $GF(2)[x]$), the FLINT (fastest), and the `Modpn` library.
- For sufficiently large input data, our CUDA code runs 20**X** to 30**X** faster.

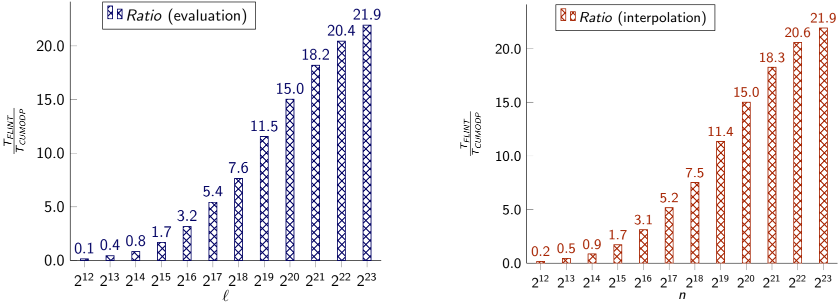


Figure: Ratio (FLINT/CUMODP) diagram for polynomials of degree $n \in \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}\}$, over $\mathbb{Z}/p\mathbb{Z}$ for a 32-bit Fourier prime p; tested on NVIDIA Tesla C2050

Outline

Outline

Bivariate solver

- Based on the theory of *regular chains* (with coefficients in small prime fields)
- Mostly written in CUDA, top level algorithm written in C, integrated into MAPLE 18 and called by `Triangularize`.
- Polynomial subresultant chains (via 2-D FFTs or subproduct-trees) and univariate polynomial GCDs are all computed in CUDA
- Experimentally, GCD computations take about 90% of the overall running, due to the CPU-GPU interactions.
- The GCD calculations use the plain algorithm since the degrees of the input polynomials are not large enough for using the FFT-based algorithm.
- The new feature of CUDA platform, called *dynamic parallelism*, should allow to push all the computations on the GPU side and substantially reduce the overheads mentioned above.

- Random input systems of dense and sparse polynomials, (the number indicates the total degree of each polynomial in system.)
- For each system, the total number of solutions is essentially the square of that degree.

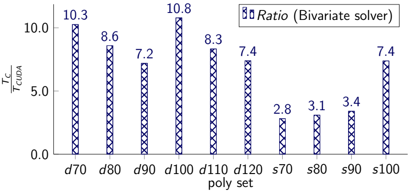


Figure: Ratio (Pure C/CUDA) diagram for testing bivariate solver on randomly generated polynomial sets

Conclusion

- For a complex application like a polynomial system solver, a CUDA implementation of its efficiency-critical subroutines can provide substantial benefit w.r.t. a pure C implementation.
- Further improvements needed: Writing the top-level algorithm in CUDA and pushing more execution on the GPU side.

Thank You!

Your Questions?

Appendix