Polynomial Computation on GPU

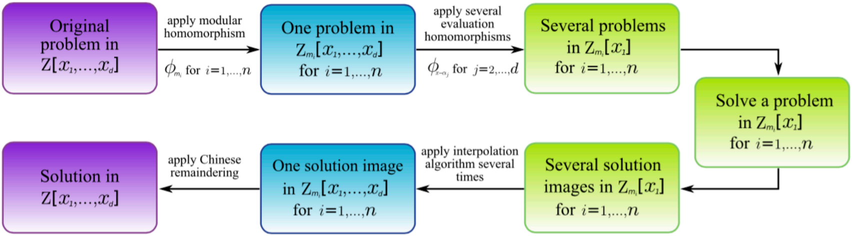
Kevin Mueller

April 22, 2016

Outline

- Overview of Process
- Setup
- Homomorphisms (Mod Reduction)
- Interpolation
- Evaluation (Kernel)
- Problems encountered
- Extensions (Multivariate)
- Demo

Overview of Process



Initial problem Setup

Suppose we have two polynomials $a(x) = 7x + 5, b(x) = 2x - 3$ and we wish to multiply them such that

$$c(x) = a(x)b(x)$$

We first need to compute bounds for our modular design

- Compute upper bound M such that our set of moduli $\prod m_i \geq 2M$
- For multiplication of two polynomials this is M $= 2||a||_\infty ||b||_\infty$

$$(7)(3)(2)(2) = 84$$

  Therefore we can take $m = 3, 5, 7$. However for this problem we will use $m = 5, 7$ for simplicity.

- Compute maximum degree for result polynomial

$$\deg(c) = \deg(a) + \deg(b)$$

Homomorphisms

There are two types of homomorphisms we will be dealing with:

- Moduluar: $\Phi_m : \mathbf{Z}[x_1, \ldots, x_v] \rightarrow \mathbf{Z}_m[x_1, \ldots, x_v]$
- In practice this simply means to take the mod on all the coefficients in the polynomial for some prime m_i. Such as

$$a_5(x) = 2x + 0 \quad b_5(x) = 2x + 2$$

- Evaluation: $\Phi_{x_i - \alpha} : \mathbf{D}[x_1, \ldots, x_v] \rightarrow \mathbf{D}[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_v]$
- This implies to evaluate each modular polynomial at each evaluation point. For a few examples:

$$a_5(0) = 0 \quad b_5(0) = 2$$

Interpolation

We can solve the interpolation problem with the vandermonde matrix **V** as

$$\mathbf{Va} = \mathbf{b}$$

where n is the number of evaluation points, **V** is the Vandermonde matrix with (i, j)-th entry $\alpha_i^j (i, j = 0, 1, \ldots, n)$, **b** is the vector with the i-th entry $b_i (i = 0, 1, \ldots, n)$, and **a** is the vector of unknown coefficients $a_i (i = 0, 1, \ldots, n)$.

$$\begin{pmatrix} c_5(0) = 0 \bmod 5 \\ c_5(1) = 3 \bmod 5 \\ c_5(2) = 4 \bmod 5 \end{pmatrix} \rightarrow c_5(x) = 4x^2 + 4x$$

$$\begin{pmatrix} c_7(0) = 6 \bmod 7 \\ c_7(1) = 2 \bmod 7 \\ c_7(2) = 5 \bmod 7 \end{pmatrix} \rightarrow c_7(x) = 3x + 6$$

Chinese Remaindering

For a set of residues $r_i \in \mathbb{Z}(1 \leq i \leq k)$ and a set of positive prime moduli $m_i \in \mathbb{Z}_{m_i}(1 \leq i \leq k)$, we can compute a unique $u \in \mathbb{Z}_m$ in the mixed radix form as

$$u = v_1 + v_2(m_1) + v(m_0 m_1) + \cdots + v_n \left(\prod_{i=0}^{k-1} m_i \right)$$

Which we can rewrite the 2nd coefficient as $M_1 = 1, M_i = m_1 m_2 \ldots m_{i-1}(i = 2, \ldots, k)$. Satisfying the system of congruences gives us

$$\begin{aligned}
v_1 &= r_1 \\
v_2 &= (r2 - v_1)\gamma_2 \mod m_2 \\
v_i &= ((r_i - v_1)\gamma_i - (\gamma_2 M_2 c_i \mod m_i) - \cdots - (v_{i-1} M_{i-1} \gamma_i \mod m_i))
\end{aligned}$$

Chinese Remaindering continued

In practice we will precompute the coefficients as

$$\gamma_i = (m_1 m_2 \dots m_{i-1})^{-1} \mod m_i$$

.

Finally we can recover the full integer r using the recursion

$$r = \gamma_1 + m_1(\gamma_2 + m_2(\gamma_3 + m_3(\dots)))$$

.

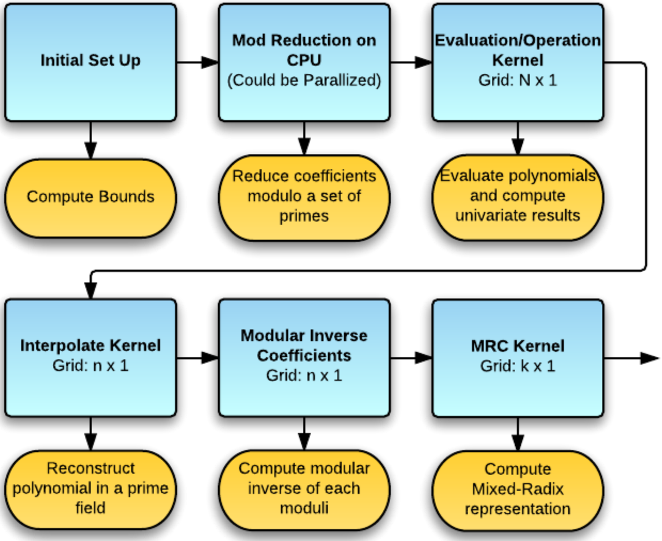


Figure: k: Number of Moduli, n: Number of Evaluation Points

Evaluation Kernel Implementation

After performing all the modular homomorphisms necessary we want to maximize threads used to evaluate the polynomials in field $\mathbf{Z}_{m_i}$. In order to accomplish this we will need to vectorize all the coefficients and perform detailed bookkeeping to correctly index this coefficient vector X. We can then find the resulting vector $Y \in \mathbb{Z}^N$ as

$$Y_i = a_j \alpha^\theta$$

,

Where $\theta = (i \mod d)$, $j = \theta + d \cdot \text{quo}(i, n/cdotd)$ and $\alpha = \text{quo}\left((i \mod nd), d\right)$ when

- l is the number of polynomials (i.e 2)
- $k = \text{len}(m)$
- $n = \deg(c) + 1$
- $d = \max\left\{\deg(a(x)), \deg(b(x))\right\}$
- $N = l \cdot k \cdot n \cdot d$

Problems Encountered

- Filling in zeros for coefficients with missing terms and handling string representations. Python libraries had poor support for this
- Python CUDA wrapper/compiler lacks some low level functionality.
- Coefficient swell in newton interpolation.
- Ill-conditioned Vandermonde matrix.
- Potential trade off between different polynomial representations.
- Sparsity required for operations in polynomial domain.
- Operation kernels for more complicated operations (division,GCD,resultant,etc).

Extensions

There is a natural tendency to want to extend the above schemes into the multivariate domain. This seems somewhat straightforward since,

$$\mathbf{D}[x_3] \xrightarrow{\Phi_{x_2-\alpha}} \mathbf{D}[x_2, x_3] \xrightarrow{\Phi_{x_1-\alpha}} \mathbf{D}[x_1, x_2, x_3]$$

That is, each multivariate problem can be decomposed into a set of univariate interpolation problems where one evaluation homomorphism $\Phi_{x_i-\alpha}$ is inverted at a time. However the interpolation becomes slightly less straight forward and sparsity becomes an even bigger issue.

A few other possible extensions could be:

- Perform the mod reductions on GPU
- Implement a parallel newton interpolation kernel
- Better support for mathematica

References

Keith Geddes
Algorithms For Computer Algebra

Pavel Emeliyanenko
Harnessing the Power of GPUs for Problems in Real Algebraic Geometry.
PhD thesis, Universitat des Saarlandes, 2012