— From Multicore to Manycore Architectures: The Reduction of Parallelization Overheads and its Impact on Implementing Polynomial Arithmetic —

ORCCA Ontario Research Centre for Computer Algebra

Anisul Sardar Haque and Marc Moreno Maza

Western

Parallelization Overheads

On multicore architectures, several phenomena (memory wall, true/false sharing, scheduling costs, etc.) limit the performances of applications which, theoretically, have a lot of opportunities for concurrent execution.

One infamous example is *Fast Fourier Transforms (FFT)*. For this type of calculation, not only the memory access pattern, but also the for-loop parallelization overheads restrict linear speedup to input vectors of very large sizes, say 2^{20}. As a consequence, on multicores, applications that depend on FFT, generally rely on serial code for these calculations.

Graphics processing units (GPUs) offer a much higher level of concurrent memory accesses. Moreover, thread scheduling is done by the hardware, which reduces for-loop parallelization overheads significantly.

In this work, we show that these hardware considerations change the view on what is fast and what is not-so-fast in polynomial arithmetic.

Polynomial Arithmetic

Asymptotically fast algorithms for polynomial arithmetic rely on FFT techniques. For univariate polynomials of degree n, these algorithms perform multiplication, division and GCD computation in $O(n\log(n)\log(\log(n)))$ arithmetic operations, while classical (or plain) algorithms, such as the Euclidean Algorithms, usually require $O(n^2)$.

Nevertheless, plain algorithms remain of interest as they are much easier to implement efficiently and as they are often faster for polynomials of small degree, say less than 2^6 to 2^9, for serial CPU code.

In this work, we show that GPU implementation (with CUDA) of plain algorithms can outperform their CPU implementation counterparts based on FFT techniques, for fairly large degrees.

The Plain Division on the GPU

Consider two univariate polynomials over a finite field

$$a = a_m x^m + \cdots + a_1 x + a_0 \text{ and } b = b_n x^n + \cdots + b_1 x + b_0, \text{ with } m \geq n.$$

The only opportunity for concurrent execution is within each division step. With the above notations, the first division step computes

$$a' \leftarrow a - \frac{a_m}{b_n} x^{m-n} b,$$

which can be viewed as a *Gaussian elimination step*. Assuming that this is done by several thread blocks, the next division step requires to broadcast the leading coefficient of the intermediate remainder a' to all thread blocks, which is a severe performance bottleneck.

Our solution consists of letting each thread block computes the leading coefficients of s consecutive intermediate remainder, for a well chosen integer s. In this way, each thread block computes a coefficient segment (of size $2\,s$) of s consecutive intermediate remainders without synchronization. Though this increases the total work by (at most) a $\frac{1}{3}$ factor, this improves performances significantly.

When sufficiently many streaming multiprocessors are available, the expected running time is $0(m-n)$, which is confirmed experimentally.

The Euclidean Algorithm on the GPU

The best parallel version of the Euclidean Algorithm which is work-efficient, is that for systolic arrays, (a model proposed by H. T. Kung and Charles E. Leiserson in 1974) for which the span is linear. However, multiprocessors based on systolic arrays are not so common.

As for the plain division, we let each thread block works on s consecutive division steps by computing the leading coefficient of the intermediate dividends. However, divisor and dividend may exchange their roles after each division step, so the algorithm is slightly more complex.

When sufficiently many streaming multiprocessors are available, the expected running time is $0(m)$, which is confirmed experimentally.

degree	GPU Euclidean	FFT-based CPU multiplication
1000	0.0104428	0.004
2000	0.0247671	0.024001
4000	0.0474535	0.056003
6000	0.0565259	0.128008
10000	0.0796034	0.200013

In the above table, GPU-implemented Euclidean Algorithm and FFT-based serial GCD computation re run on the same desktop for univariate polynomials modulo a 30-bit prime.

Plain Polynomial Multiplication on the GPU

We parallelize the computation of the product $a \times b$ based on the plain algorithm that we have all learned it in primary school for integers.

- Multiplication phase: The set of terms $a_i b_j x^{i+j}$ forms a trapezoid that we decompose into smaller trapezoids such that each of them can be computed by a thread block. Within a thread block, a thread computes all the terms of a given degree and adds them up into a vector coefficient, where this *vector* is associated to the thread block.
- Addition phase: All the thread block vectors are added together by means of a parallel reduction.

When sufficiently many streaming multiprocessors are available, the expected running time is $0(\log(m))$, which is confirmed experimentally.

degree	GPU Plain multiplication	GPU FFT-based multiplication
2^{10}	0.00049	0.0044136
2^{11}	0.0009	0.004642912
2^{12}	0.0032	0.00543696
2^{13}	0.01	0.00543696
2^{14}	0.045	0.00709072

In the above table, plain and FFT-based multiplication (both highly optimized CUDA codes) are run on a Tesla 2050 for univariate polynomials modulo a 30-bit prime.

Acknowledgments

NSERC CRSNG

Maplesoft Mathematics • Modeling • Simulation

MITACS