On the Parallelization of Subproduct Tree Techniques Targeting Many-core Architectures

Sardar Anisul Haque, Farnam Mansouri, Marc Moreno Maza

University of Western Ontario, London, Ontario, Canada
haque.sardar@gmail.com, mansouri.farnam@gmail.com, moreno@csd.uwo.ca

Abstract. We propose parallel algorithms for operations on univariate polynomials (multi-point evaluation, interpolation) based on subproduct tree techniques and targeting many-core GPUs. On those architectures, we demonstrate the importance of adaptive algorithms, in particular the combination of parallel plain arithmetic and parallel FFT-based arithmetic. Experimental results illustrate the benefits of our algorithms.

1 Introduction

We investigate the use of Graphics Processing Units (GPUs) in the problems of evaluating and interpolating polynomials. Many-core GPU architectures were considered in [17] and [18] in the case of numerical computations, with the purpose of obtaining better support, in terms of accuracy and running times, for the development of polynomial system solvers.

Our motivation, in this work, is also to improve the performance of polynomial system solvers. However, we are targeting symbolic, thus exact, computations. In particular, we aim at providing GPU support for solvers of polynomial systems with coefficients in finite fields, such as the one presented in [14]. This case handles problems from cryptography and serves as a base case for the so-called modular methods [4], since those methods reduce computations with rational number coefficients to computations with finite field coefficients.

Finite fields allow the use of asymptotically fast algorithms for polynomial arithmetic, based on Fast Fourier Transforms (FFTs) or, more generally, subproduct tree techniques[1], which have the advantage of providing a more general setting than FFTs. More precisely, evaluation points do not need to be successive powers of a primitive root of unity. Evaluation and interpolation based on subproduct tree techniques have "essentially" (up to log factors) the same algebraic complexity estimates as their FFT-based counterparts. However, their implementation is known to be challenging.

In this work, we report on the first GPU implementation (using CUDA [16]) of subproduct tree techniques for multi-point evaluation and interpolation of univariate polynomials. The parallelization of those techniques raises the following challenges on hardware accelerators:

[1] Chapter 10 of [5] and the paper [1] contain overviews of those techniques.

1. The divide-and-conquer formulation of operations on subproduct-trees is not sufficient to provide enough parallelism and one must also parallelize the underlying polynomial arithmetic operations, in particular multiplication.
2. Algorithms based on FFT (such as subproduct tree techniques) are memory bound since the ratio of work to memory access is essentially constant, which makes those algorithms not well suited for multi-core architectures.
3. During the course of the execution of a subproduct tree operation (construction, evaluation, interpolation) the degrees of the involved polynomials vary greatly, thus so does the work load of the tasks, which makes those algorithms complex to implement on many-core GPUs.

The contributions of this work are summarized below. We propose parallel algorithms for performing subproduct tree construction, evaluation and interpolation. We also report on their implementation on many-core GPUs. See Sections 3, 5 and 6, respectively. We enhance the traditional algorithms for polynomial evaluation and interpolation based on subproduct tree techniques, by introducing the data-structure of a *subinverse tree*, which we use to implement both evaluation and interpolation, see Section 4. For subproduct tree operations targeting many-core GPUs, we demonstrate the importance of *adaptive algorithms*[2] That is, algorithms that adapt their behavior according to the available computing resources. In particular, we combine *parallel plain arithmetic* and *parallel fast arithmetic*. For the former we rely on [7] and, for the latter we extend the work of [13]. The span and parallelism overhead of our algorithm are measured considering the *many-core machine model* of [8]. The paper [15] briefly discusses the parallelization of FFT-based multi-point evaluation without considering parallelism overhead, adaptive algorithms nor reporting on an implementation.

To evaluate our implementation, we measure the effective memory bandwidth of our GPU code for parallel multi-point evaluation and interpolation on a card with a theoretical maximum memory bandwidth of 148 GB/S, our code reaches peaks at 64 GB/S. Since the arithmetic intensity of our algorithms is high, we believe that this is a promising result.

All implementation of subproduct tree techniques that we are aware of are serial only. This includes [3] for $GF(2)[x]$, the FLINT library[9] and the `Modpn` library [10]. Hence we compare our code against probably the best serial C code (the FLINT library) for the same operations. For sufficiently large input data and on NVIDIA Tesla C2050, our code outperforms its serial counterpart by a factor ranging between 20 to 30. Experimental data are provided in Section 7. Our code is freely available in source, under GPL license, as part of the project *CUDA Modular Polynomial* (CUMODP) whose web site is `http://www.cumodp.org`.

2 Background

We refer to [16] for notions related to GPU programming. We review below the notion of a subproduct tree and specify costs for the underlying polynomial

[2] A famous example of adaptive algorithm usage was for computing 2,700 billion decimal digits of π on a desktop computer by F. Bellard `http://bellard.org/pi/`.

arithmetic used in our implementation. Notations and hypotheses introduced in this section are used throughout this paper. Let $n = 2^k$ for some positive integer k and let $\mathbb{K}$ be a finite field. Let $u_0, \ldots, u_{n-1} \in \mathbb{K}$. Define $m_i = x - u_i$, for $0 \le i < n$. We assume that each $u_i \in \mathbb{K}$ can be stored in one machine word.

Subproduct tree. The subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$ is a complete binary tree of height $k = \log_2 n$. The j-th node of the i-th level of M_n is denoted by $M_{i,j}$, where $0 \le i \le k$ and $0 \le j < 2^{k-i}$, and is defined by $M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \cdots m_{j \cdot 2^i + (2^i - 1)} = \prod_{0 \le \ell < 2^i} m_{j \cdot 2^i + \ell}$. Each $M_{i,j}$ can be defined recursively by $M_{0,j} = m_j$ and $M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}$. The i-th level of M_n has 2^{k-i} polynomials with degree of 2^i. Since each element of $\mathbb{K}$ fits a machine word, storing the subproduct tree M_n requires at most $n \log_2 n + 3n - 1$ words.

Algorithm 1: $\mathsf{SubproductTree}(m_0, \ldots, m_{n-1})$

Input: $m_0 = (x - u_0), \ldots, m_{n-1} = (x - u_{n-1}) \in \mathbb{K}[x]$ with $u_i \in \mathbb{K}$, $n = 2^k$, $k \in \mathbb{N}$.
Output: The subproduct-tree M_n.

```
for j = 0 to n − 1 do
    M_{0,j} = m_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        M_{i,j} = M_{i−1,2j} M_{i−1,2j+1};
return M_n;
```

Multi-point evaluation and interpolation. Given a univariate polynomial $f \in \mathbb{K}[x]$ of degree less than n, we define $\chi(f) = (f(u_0), \ldots, f(u_{n-1}))$. The map χ is called the *multi-point evaluation map* at $u_0, \ldots, u_{n-1}$. When $u_0, \ldots, u_{n-1}$ are pairwise distinct, then it realizes an isomorphism of $\mathbb{K}$-vector spaces $\mathbb{K}[x]/\langle m \rangle$ and $\mathbb{K}^n$, where $m = \prod_{0 \le i < n}(x - u_i)$. The inverse map χ^{-1} can be computed via Lagrange interpolation. Given values $v_0, \ldots, v_{n-1} \in \mathbb{K}$, the unique polynomial $f \in \mathbb{K}[x]$ of degree less than n which takes the value v_i at the point u_i for all $0 \le i < n$ is: $f = \sum_{i=0}^{n-1} v_i s_i m/(x - u_i)$ where $s_i = \prod_{i \ne j,\, 0 \le j < n} 1/(u_i - u_j)$.

Complexity measures. Since we are targeting GPU implementation, our parallel algorithms are analyzed using an appropriate model of computation introduced in [8]. The complexity measures are the *work* (i.e. algebraic complexity estimate) the *span* (i.e. running time on infinitely many processors) and the *parallelism overhead*. This latter is the total time for transferring data between the global memory and the local memories of the streaming multi-processors (SMs).

Plain multiplication. The number of arithmetic operations for multiplying two polynomials with degree less than d using the *plain (schoolbook) multiplication* is $\mathsf{M}_{\mathrm{plain}}(d) = 2d^2 - 2d + 1$. In our GPU implementation, when d is small enough, each polynomial product is computed by a single thread-block and thus within the local memory of a single SM. In this case, we use $2d+2$ threads for one polynomial

multiplication. Each thread copies one coefficient from global memory to the local memory. Each of these threads, except one, is responsible for computing one coefficient of the output polynomial and writes that coefficient back to global memory. So the span and parallelism overhead are $d+1$ and $2U$ respectively, where $1/U$ is the throughput measured in word per second, see [8].

FFT-based multiplication. The number of operations for multiplying two polynomials with degree less than d using *Cooley-Tukey's FFT* algorithms is $\mathsf{M}_{\mathrm{FFT}}(d) = 9/2\ d^{\prec}\log_2(d^{\prec}) + 4d^{\prec}$ [11]. Here $d^{\prec} = 2^{\lceil \log_2(2d-1)\rceil}$. In our GPU implementation, which relies on Stockham FFT algorithm, this number of operations becomes: $\mathsf{M}_{\mathrm{FFT}}(d) = 15d^{\prec}\log_2(d^{\prec}) + 2d^{\prec}$, see [13]. The span and parallelism overhead of our FFT-based multiplication are $15d^{\prec} + 2d^{\prec}$ and $(36d^{\prec} - 21)U$ respectively.

Polynomial division. Given $a, b \in \mathbb{K}[x]$, with $\deg(a) \geq \deg(b)$ we denote by $\mathsf{Remainder}(a, b)$ the remainder in the *Euclidean division* of a by b. The number of arithmetic operations for computing $\mathsf{Remainder}(a, b)$, by plain division, is $(\deg(b)+1)(\deg(a)-\deg(b)+1)$. In our GPU implementation, we perform plain division for small degree polynomials, in which case a, b are stored into the local memory of an SM. For larger polynomials, we use an FFT-based algorithm to be discussed later. Returning to plain division, we use $\deg(b)+1$ threads to implement this operation. Each thread reads one coefficient of b and at most $\lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil$ coefficients of a from the global memory. For the output, at most $\deg(b)$ threads write the coefficients of the remainder to the global memory. The span and parallelism overhead are $2(\deg(a)-\deg(b)+1)$ and $(2+\lceil \frac{\deg(a)+1}{\deg(b)+1} \rceil)U$.

Reversal of a polynomial. For $f \in \mathbb{K}[x]$ of degree $d > 0$ and for $e \geq d$, the *reversal* of order e of f is the polynomial denoted by $\mathsf{rev}_e(f)$ and defined as $\mathsf{rev}_e(f) = x^e f(1/x)$. In our implementation, we use one thread for each coefficient of the input and output. So the span and overhead are 1 and $2U$, respectively.

Inverse modulo a power of x. For $f \in \mathbb{K}[x]$, with $f(0) = 1$, and $\ell \in \mathbb{N}$ the *modular inverse* of f modulo x^ℓ is denoted by $\mathsf{Inverse}(f, \ell)$ and is uniquely defined by $\mathsf{Inverse}(f, \ell)\, f \equiv 1 \mod (x^\ell)$. One can compute $\mathsf{Inverse}(f, \ell)$ by Newton iteration, see [5, Chapter 10] for details in sequential time $O(\mathsf{M}_{\mathrm{FFT}}(\ell))$.

To help the reader following the complexity analysis presented in the sequel of this paper, a Maple worksheet can be found at `http://cumodp.org/links.html`. It provides estimates for space allocation, work (total of number of arithmetic operations), span (parallel running time) and parallelism overhead for constructing subproduct tree and sub-inverse tree (our proposed data structure). Recall that the parallelism overhead measures the time for transferring data between the device global memory and the SMs' shared memories. The estimates that we provide follow our CUDA implementation available at `http://cumodp.org`.

3 Subproduct tree construction

We propose an adaptive algorithm for constructing the subproduct tree $M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$. We fix an integer H with $1 \leq H \leq k$. We call the following procedure an *adaptive algorithm for computing M_n with threshold H*:

1. for each level h, with $1 \le h \le H$, nodes are computed via plain multiplication,
2. for each level h, with $H+1 \le h \le k$, nodes are computed via FFT-based multiplication.

This algorithm is adaptive in the sense that it takes into account the amount of available resources, as well as the input data size. Indeed, as specified in Section 2, each plain multiplication is performed by a single SM, while each FFT-based multiplication is computed by a kernel call, thus using several SMs. In fact, this kernel computes a number of FFT-based products concurrently.

Before analyzing this adaptive algorithm, we recall that, if the subproduct tree M_n is computed by means of a single multiplication algorithm, with multiplication time[3] $\mathsf{M}(\mathsf{n})$, Lemma 10.4 in [5] states that the total number of operations for constructing M_n is at most $\mathsf{M}(n)\log_2 n$ operations in $\mathbb{K}$. We also note that the leading coefficient of each polynomial in M_n is one. Thus this coefficient does not need to be stored in memory. Moreover, this allows us to multiply two polynomials at level i, for $H+1 \le i \le k-1$, via FFTs of size 2^{i+1} (instead of 2^{i+2} with a naive approach that would ignore that leading coefficients are one).

Another implementation trick is the so-called *FFT doubling*. At a level $H+2 \le i \le k$, for $0 \le j \le 2^{k-i}-1$, consider how to compute $M_{i,j}$ from $M_{i-1,2j}$ and $M_{i-1,2j+1}$. Since the values of $M_{i-1,2j}$ and $M_{i-1,2j+1}$ at 2^{i-1} points have already been computed (via FFT), it is sufficient, in order to determine $M_{i,j}$, to evaluate $M_{i-1,2j}$ and $M_{i-1,2j+1}$ at 2^{i-1} additional points. To do this, we write $f \in \{M_{i-1,2j}, M_{i-1,2j+1}\}$ as $f = f_0 + x^{2^{i-2}} f_1$, with $\deg(f_0) < 2^{i-2}$, and evaluate each of f_0, f_1 at those 2^{i-1} additional points. While this trick brings savings in terms of work, it increases memory footprint, in particular parallelism overheads. Integrating this trick in our implementation is work in progress and, in the rest of this paper, the theoretical and experimental results do not rely on it.

Proposition 1 *The number of arithmetic operations of the adaptive algorithm for computing M_n with threshold H is*

$$n\left(\frac{15}{2}\log_2(n)^2 + \frac{19}{2}\log_2(n) + 2^H - \frac{15}{2}H^2 - \frac{17}{2}H - \frac{1}{2^H}\right).$$

Proposition 2 *The number of machine words required for storing M_n, with threshold H is given below*

$$n\left(\log_2(n) - H + 5\right) + (-H-2)\left(n + \frac{n}{2^{H+1}}\right) + 2nH\left(1 + \frac{1}{2^{H+2}}\right)$$

Proposition 3 *Span and overhead for constructing M_n with threshold H using our adaptive method are span_{M_n} and $\mathsf{overhead}_{M_n}$ respectively, where*

$$\mathsf{span}_{M_n} = \frac{15}{2}\left(\log_2(n)+1\right)^2 - \frac{7}{2}\log_2(n) + 2^{H+1} - \frac{15}{2}\left(H+1\right)^2 + \frac{9}{2}H - 2$$

and

$$\mathsf{overhead}_{M_n} = \left(\left(18\left(\log_2(n)+1\right)^2 - 35\log_2(n) - 18\left(H+1\right)^2 + 35H\right) + 2H\right)U.$$

[3] This notion is defined in [5, Chapter 8]

The proof of Propositions 1, 2 and 3 are based on the hypotheses stated in Section 2 and elementary calculations, which, to the interest of space, can be found in our MAPLE worksheet at `http://cumodp.org/links.html`.

Propositions 1 and 3 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is very satisfactory. We stress the fact that this result could be achieved because both our plain and FFT-based multiplications are parallelized. Observe also that, for a fixed n, parallelism overhead decreases as H increases: that is, plain multiplication suffers less parallelism overheads than FFT-based multiplication on GPUs.

It is natural to ask how to choose H so as to minimize work and span. Elementary calculations, using our MAPLE worksheet suggest $6 \leq H \leq 7$. However, in degrees 2^6 and 2^7, parallelism overhead is too high for FFT-based multiplication and, experimentally, the best choice appeared to be $H = 8$.

4 Subinverse tree construction

For $f \in \mathbb{K}[x]$ of degree less than n, evaluating f on the point set $\{u_0, \ldots, u_{n-1}\}$ is done by Algorithm 2 by calling $\mathsf{TopDownTraverse}(f, k, 0, M_n, F)$. An array F of length n is passed to this procedure such that $F[i]$ receives $f(u_i)$ for $0 \leq i \leq n-1$. The function call $\mathsf{Remainder}(f, M_{i,j})$ relies on plain division whenever $i < H$ holds, where H is the threshold of Section 3. Fast division is applied when

Algorithm 2: $\mathsf{TopDownTraverse}(f, i, j, M_n, F)$

Input: $f \in \mathbb{K}[x]$ with $\deg(f) < 2^i$, i and j are integers such that $0 \leq i \leq k$, $0 \leq j < 2^{k-i}$ and F is an array of length n.

```
if i == 0 then
    F[j] = f;
    return;
f0 = Remainder(f, M_{i-1,2j});
f1 = Remainder(f, M_{i-1,2j+1});
TopDownTraverse(f0, i - 1, 2j, M_n, F);
TopDownTraverse(f1, i - 1, 2j + 1, M_n, F);
```

polynomials are large enough and, actually, can not be stored within the local memory of a streaming multiprocessor.

Fast division requires computing $\mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$, for $H \leq i \leq k$ and $0 \leq j < 2^{k-i}$, see Chapter 9 in [5]. However, this latter calculation has, in principle, to be done via Newton iteration. As mentioned in Section 2, this latter provides little opportunities for concurrent execution. To overcome this performance issue, we introduce a strategy that relies on a new data structure called *subinverse tree*. In this section, we first define subinverse trees and describe their implementation. Then, we analyze the complexity of constructing a subinverse tree.

Definition 1 *For the subproduct tree* $M_n \coloneqq \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$*, the* subinverse tree *associated with* M_n*, denoted by* InvM_n*, is a complete binary tree of the same format as* M_n*, defined as follows. For* $0 \le i \le k$*, for* $0 \le j < 2^{k-i}$*, the* j*-th node of level* i *in* InvM_n *contains the univariate polynomial* $\mathsf{InvM}_{i,j}$ *of less than degree* 2^i *and defined by*

$$\mathsf{InvM}_{i,j}\,\mathsf{rev}_{2^i}(M_{i,j}) \;\equiv\; 1 \mod x^{2^i}.$$

Note that we do not store the polynomials of the subinverse tree InvM_n below level H. Indeed, for those levels, we rely on plain division for the function calls $\mathsf{Remainder}(f, M_{i,j})$ in Algorithm 2.

Proposition 4 *Let* InvM_n *be the subinverse tree associated with the subproduct tree* M_n*, with the threshold* $H < k$*. Then, the amount of space required for storing* InvM_n*, is* $(k - H)n$*.*

The following lemma is a simple observation from which we derive Proposition 5 and, thus, the principle of subinverse tree construction.

Lemma 1 *Let* $\mathbb{R}$ *be a commutative ring with identity element. Let* $a, b, c \in \mathbb{R}[x]$ *be univariate polynomials such that* $c = a\,b$ *and* $a(0) = b(0) = 1$ *hold. Let* $d = \deg(c) + 1$*. Then, we have* $c(0) = 1$ *and* $\mathsf{Inverse}(c, d) \mod x^d$ *can be computed from* a *and* b *as follows:* $\mathsf{Inverse}(c, d) \;\equiv\; \mathsf{Inverse}(a, d) \cdot \mathsf{Inverse}(b, d) \mod x^d$.

Proposition 5 *Let* $\mathsf{InvM}_{i,j}$ *be the* j^{th} *polynomial (from left to right) of the subinverse tree at level* i*, where* $0 < i < k$ *and* $0 \le j < 2^{k-i}$*. We have:*

$$\mathsf{InvM}_{i,j} \equiv \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i) \cdot \mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i) \mod x^{2^i}$$

where $\mathsf{InvM}_{i,j} = \mathsf{Inverse}(\mathsf{rev}_{2^i}(M_{i,j}), 2^i)$ *from Definition 1.*

We observe that computing $\mathsf{InvM}_{i,j}$ requires $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^i)$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^i)$. However, at level $i-1$, the nodes $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ are $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), 2^{i-1})$ and $\mathsf{Inverse}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), 2^{i-1})$ respectively. To take advantage of this observation, we call $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j}), \mathsf{InvM}_{i-1,2j}, i-1)$ and $\mathsf{OneStepNewtonIteration}(\mathsf{rev}_{2^{i-1}}(M_{i-1,2j+1}), \mathsf{InvM}_{i-1,2j+1}, i-1)$, see Algorithm 3, so as to obtain $\mathsf{Inverse}(M_{i-1,2j}, 2^i)$ and $\mathsf{Inverse}(M_{i-1,2j+1}, 2^i)$ respectively. Algorithm 3 performs a single iteration of *Newton iteration*'s algorithm. Finally, we perform one truncated polynomial multiplication, as stated in Proposition 5, to obtain $\mathsf{InvM}_{i,j}$. We apply this technique to compute all the polynomials of level i of the subinverse tree, for $H + 1 \le i \le k$.

Since we do not store the leading coefficients of the polynomials in the subproduct tree, our implementation relies on a modified version of Algorithm 3, namely Algorithm 4.

Let $f = \mathsf{rev}_{2^i}(M_{i,j})$ and $g = \mathsf{InvM}_{i,j}$. From Definition 1, we have $fg \equiv 1 \bmod x^{2^i}$. Note that $\deg(fg) \le 2^{i+1} - 1$ holds. Let $e^{\prec} = -fg + 1$. Thus $e^{\prec}$ is a polynomial of degree at most $2^{i+1} - 1$. Moreover, from the definition of a subinverse tree, we know its least significant 2^i coefficients are zeros. Let $e = e^{\prec}/x^{2^i}$.

Algorithm 3: OneStepNewtonIteration(f, g, i)

Input: $f, g \in \mathbb{R}[x]$ such that $f(0) = 1$, where $\deg(g) =\leq 2^i$ and $fg \equiv 1 \mod x^{2^i}$.
Output: $g^{\prec} \in \mathbb{R}[x]$ such that $fg^{\prec} \equiv 1 \mod x^{2^{i+1}}$.
$g^{\prec} = (2g - fg^2) \mod x^{2^{i+1}}$;
return $g^{\prec}$;

So $\deg(e) \leq 2^i - 1$. In Algorithm 3, we have $g^{\prec} \equiv g \mod x^{2^i}$. We can compute $g^{\prec}$ from eg and g. The advantage of working with e instead of $e^{\prec}$ is that the degree of $e^{\prec}$ is twice the degree of e. In Algorithm 4, we compute e as $e = -\mathsf{rev}_{2^i}(M_{i,j} \cdot \mathsf{rev}_{2^i-1}(\mathsf{InvM}_{i,j}) - x^{2^{i+1}-1})$.

Algorithm 4: EfficientOneStep$(M_{i,j}^{\prec}, \mathsf{InvM}_{i,j}, i)$

Input: $M_{i,j}^{\prec} = M_{i,j} - x^{2^i}$, $\mathsf{InvM}_{i,j}$.
Output: g, such that $g \ \mathsf{rev}_{2^i}(M_{i,j}) \equiv 1 \mod x^{2^{i+1}}$.
$a =\mathsf{rev}_{2^i-1}(\mathsf{InvM}_{i,j})$;
$b = a - x^{2^i-1}$;
$c =\mathsf{convolution}(a, M_{i,j}^{\prec}, 2^i)$;
$d =\mathsf{rev}_{2^i}(c + b)$;
$e = -d$;
$h = e \ \mathsf{InvM}_{i,j} \mod x^{2^i}$;
$g = hx^{2^i} +\mathsf{InvM}_{i,j}$;
return g;

The *Middle product* technique [6] is used in Algorithm 3 for computing c.

For a given i, with $H < i \leq k$, and for $0 \leq j < 2^{k-i}$, Algorithm 5 computes the polynomial $\mathsf{InvM}_{i,j}$. Algorithm 5 calls Algorithm 4 twice to increase the accuracy of $\mathsf{InvM}_{i-1,2j}$ and $\mathsf{InvM}_{i-1,2j+1}$ to x^{2i}. Then it multiplies those latter polynomials and applies a mod operation. Algorithm 6 is the top level algorithm which creates the subinverse tree InvM_n using a bottom-up approach and calling Algorithm 5 for computing each node $\mathsf{InvM}_{i,j}$ for $H \leq i \leq k$ and $0 \leq j < 2^{k-i}$.

Propositions 6 and 7 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is satisfactory.

Proposition 6 *For the subproduct tree M_n, with threshold H, the number of arithmetic operations for constructing the subinverse tree InvM_n using Algorithm 6 is:*

$$n\left(10\left(3\log_2(n)^2 + \log_2(n) - 3H^2 - 7H - 4\right) + \frac{16\,4^{2^H}}{3 \cdot 2^H} + 2 - \frac{1}{3 \cdot 2^H} - \frac{2}{2^{H-2^H}}\right).$$

Algorithm 5: InvPolyCompute(M_n,InvM, i, j)

Input: M_n and InvM are the subproduct tree and subinverse tree respectively.
Output: c such that $c \ \mathsf{rev}_{2^i}(M_{i,j}) \equiv 1 \bmod x^{2^i}$.
$M^{\prec}_{i-1,2j} = M_{i-1,2j} - x^{2^{i-1}}$;
$M^{\prec}_{i-1,2j+1} = M_{i-1,2j+1} - x^{2^{i-1}}$;
$a =$ EfficientOneStep($M^{\prec}_{i-1,2j}$,InvM$_{i-1,2j}, i-1$) ;
$b =$ EfficientOneStep($M^{\prec}_{i-1,2j+1}$,InvM$_{i-1,2j+1}, i-1$) ;
$c = ab \bmod x^{2^i}$;
return c;

Algorithm 6: SubinverseTree(M_n, H)

Input: M_n is the subproduct tree and $H \in \mathbb{N}$.
Output: the subinverse tree InvM$_n$
for $j = 0 \dots 2^{k-H} - 1$ **do**
    InvM$_{H,j}$ = Inverse($M_{H,j}, \deg(M_{H,j})$);
for $i = (H+1) \dots k$ **do**
    for $j = 0 \dots 2^{k-i} - 1$ **do**
        InvM$_{i,j}$ = InvPolyCompute(M_n,InvM$_{i,j}$);
return InvM$_n$;

Proposition 7 *For the subproduct tree M_n with threshold H, the span and overhead of constructing the subinverse tree* InvM$_n$ *by Algorithm 6 are* span$_{\mathsf{InvM}_n}$ *and* overhead$_{\mathsf{InvM}_n}$ *respectively, where*

$$\mathsf{span}_{\mathsf{InvM}_n} = \frac{75}{2}\log_2(n)^2 - \frac{107}{2}\log_2(n) + 2\cdot 4^H + 4\cdot 2^H - \frac{75}{2}H^2 - \frac{43}{2}H + 14$$

and

$$\mathsf{overhead}_{\mathsf{InvM}_n} = U\left(90\log_2(n)^2 - 255\log_2(n) + 2^{H+1} - 90H^2 + 75H + 166\right).$$

5 Polynomial evaluation

Algorithm 2 solves the multi-point evaluation problem using subproduct tree technique. To do so, we construct the subproduct tree M_n with threshold H and the corresponding subinverse tree InvM$_n$. Then, we run Algorithm 2, which requires polynomial division. We implement both plain and fast division. For the latter, we rely on the subinverse tree, as described in Section 4

Proposition 8 *For the subproduct tree M_n with threshold H and its corresponding subinverse tree* InvM$_n$*, the number of arithmetic operations of Algorithm 2 is:*

$$30\,n\log_2(n)^2 + 106\,n\log_2(n) + n\,2^{H+1} - 30\,n\,H^2 - 46\,n\,H + 74\,n + 16\,\frac{n}{2^H} - 8.$$

In [12], the algebraic complexity estimate for performing multi-point evaluation (which only considers multiplication cost and ignores other coefficient operations) is $7M(n/2)\log_2(n) + O(M(n))$. Considering for $M(n)$ a multiplication time like the one based on Cooley-Tukey's algorithm (see Section 2) the running time estimate of [12] becomes similar to the estimate of Proposition 8. Since our primary goal is paralllelization, we view this comparison as satisfactory. Furthermore, Propositions 8 and 9 imply that for a fixed a H, the parallelism (ratio work to span) is in $\Theta(n)$ which is satisfactory as well.

Proposition 9 *Given a subproduct tree M_n with threshold H and the corresponding subinverse tree InvM_n, span and overhead of Algorithm 2 are $\mathsf{span}_{\mathsf{eva}}$ and $\mathsf{overhead}_{\mathsf{eva}}$ respectively, where*

$$\mathsf{span}_{\mathsf{eva}} = 15\log_2(n)^2 + 23\log_2(n) + 6 \times 2^H - 15\,H^2 - 22\,H - 2$$

and

$$\mathsf{overhead}_{\mathsf{eva}} = \left(36\log_2(n)^2 + 3\log_2(n) - 36\,H^2 + 2\,H\right)U.$$

6 Polynomial interpolation

As recalled in Section 2, we rely on Lagrange interpolation. Our interpolation procedure, inspired by the recursive algorithm in [5, Chapter 10], relies on Algorithm 7 below, which proceeds in a bottom-up traversal fashion.

Algorithm 7: $\mathsf{LinearCombination}(M_n, c_0, \ldots, c_{n-1})$

Input: Precomputed subproduct tree M_n for the evaluation points $u_0, \ldots, u_{n-1}$, and $c_0, \ldots, c_{n-1} \in \mathbb{K}$, with $n = 2^k$ for $k \in \mathbb{N}$

Output: $\sum_{0 \le i < n} c_i m/(x - u_i) \in \mathbb{K}[x]$, where $m = \prod_{0 \le i < n}(x - u_i)$

```
for j = 0 to n − 1 do
    I_{0,j} = c_j;
for i = 1 to k do
    for j = 0 to 2^{k−i} − 1 do
        I_{i,j} = M_{i−1,2j} I_{i−1,2j+1} + M_{i−1,2j+1} I_{i−1,2j};
return I_{k,0};
```

Algorithm 7 computes a binary tree such that the j-th node from the left at level i is a polynomial $I_{i,j}$ of degree $2^i - 1$, for $0 \le i \le k$, $0 \le j \le 2^{k-i} - 1$. The root $I_{k,0}$ is the desired polynomial. We use the same threshold H as for the construction of the subproducttree tree:

1. for each node $I_{i,j}$ where $1 \le i \le H$ and $0 \le j < 2^{k-i}$, we compute $I_{h,j}$ using plain multiplication.

2. for each node $I_{i,j}$, with $H+1 \leq i \leq k$, we compute the $I_{i,j}$ using FFT-based multiplication.

In Theorem 10.10 in [5], the complexity estimate for the *Linear Combination* is $(\mathsf{M}(n)+O(n))\log(n)$. In Proposition 10, we present a more precise estimate.

Proposition 10 *For the subproduct tree M_n with threshold H, the number of arithmetic operations Algorithm 7 is given below*

$$15\,n\log_2(n)^2 + 20\,n\log_2(n) + 11\,n + 13\,n\,H - 15\,n\,H^2 + n\,2^{H+1} - n\,2^{1-H}.$$

Proposition 11 *For the subproduct tree M_n with threshold H and the corresponding subinverse tree InvM_n, the span and overhead of Algorithm 7 are $\mathsf{span}_{\mathsf{lc}}$ and $\mathsf{overhead}_{\mathsf{lc}}$ respectively, where*

$$\mathsf{span}_{\mathsf{lc}} = \frac{15}{2}\log_2(n)^2 + \frac{25}{2}\log_2(n) + 2^{H+1} - \frac{15}{2}H^2 - \frac{21}{2}H - 2$$

and

$$\mathsf{overhead}_{\mathsf{lc}} = 18\log_2(n)^2 + \log_2(n) - 18\,H^2 + 4\,H.$$

Finally we use Algorithm 8 in which we first compute $c_0, \ldots, c_{n-1}$, and then we call Algorithm 7. Algorithm 8 is adapted from Algorithm 10.11 in [5].

Algorithm 8: $\mathsf{FastInterpolation}(u_0, \ldots, u_{n-1}, v_0, \ldots, v_{n-1})$

Input: $u_0, \ldots, u_{n-1} \in \mathbb{K}$ such that $u_i - u_j$ is a unit for $i \neq j$, and $v_0, \ldots, v_{n-1} \in \mathbb{K}$, and $n = 2^k$ for $k \in \mathbb{N}$

Output: The unique polynomial $P \in \mathbb{K}[x]$ of degree less than n such that $P(u_i) = v_i$ for $0 \leq i < n$

$M_n := \mathsf{SubproductTree}(u_0, \ldots, u_{n-1})$;
Let m be the root of M_n;
Compute $m^{\prime}(x)$ the derivative of m;
$\mathsf{InvM}_n := \mathsf{SubinverseTree}(M_n, H)$;
$\mathsf{TopDownTraverse}(m^{\prime}(x), i, j, M_n, F)$;
return $\mathsf{LinearCombination}(M_n, v_0/F[0], \ldots, v_{n-1}/F[n-1])$;

From the different propositions of this paper, it follows that, for a fixed H, the parallelism (ratio work to span) of Algorithm 8 is in $\Theta(n)$ which is satisfactory.

7 Experimentation

The algorithms presented in this paper have been implemented in CUDA [16] as part of the CUMODP library. The FFT-based algorithms of this library are described [13,14] while those based on plain arithmetic are presented in [7]. As mentioned before, our FFT computations use Stockham algorithms which is known

Table 1: Effective memory bandwidth (in GB/S). The input size is $n = 2^k$.

k	Evaluation	Interpolation
11	0.2554	0.3403
12	0.5596	0.7054
13	1.2947	1.6182
14	2.5838	3.1445
15	5.2702	6.3464
16	9.6193	11.4143
17	16.4358	18.7800
18	22.6172	26.7590
19	32.3230	38.7674
20	40.4644	49.0012
21	46.7343	57.0978
22	50.8830	62.4516
23	52.9413	64.2464

Table 2: Multiplication timings (in sec.) for polynomials of size 2^k: CUMODP vs FLINT.

k	CUMODP (s)	FLINT (s)	Ratio
11	0.0019	0.002	1.029
12	0.0032	0.003	0.917
13	0.0023	0.008	3.441
14	0.0039	0.013	3.346
15	0.0032	0.023	7.216
16	0.0065	0.045	6.942
17	0.0084	0.088	10.475
18	0.0122	0.227	18.468
19	0.0198	0.471	23.738
20	0.0266	1.011	27.581
21	0.0718	2.086	29.037
22	0.1451	4.419	30.454
23	0.3043	9.043	29.717

to be more appropriate for many-core GPUs thah the one of Cooley-Tukey. We focus on radix-2 FFTs [13] and rely on an optimized version of Montgomery's trick [12] for modular multipoint [4]

We run our CUDA codes on a NVIDIA Tesla M2050 GPU card and we run the other codes on the same machine equipped with an Intel Xeon X5650 CPU at 2.67GHz. Our test cases use random points or random polynomials with coefficients in a prime field whose characteristic is a 30-bit prime number.

With Table 1 we evaluate the intrinsic quality of this implementation while with Tables 2, 3 and Figures 1, 2 we provide comparative benchmark results.

One of the major factors of performance in GPU applications is of *memory bandwidth*. For our implementation of multi-point evaluation and interpolation, this factor is presented for various input sizes in the Table 1. The maximum memory bandwidth for our GPU card is 148 GB/S. Since our code has a high arithmetic intensity, we believe that our experimental results are promising, while leaving room for improvement.

In Table 2, we compare two implementations of FFT-based polynomial multiplication. The first one is that the CUMODP library, presented [13]. The second one is from the FLINT library [9]. From the experimental data, it is clear that, our CUDA code for FFT-based multiplication outperforms its FLINT counterpart only in size larger than 2^{13}. Thus, we need to implement another multiplication algorithm to have better performance in low-to-average degrees. This is work in progress.

In Table 3 we compare our implementation of multi-point polynomial evaluation and polynomial interpolation with that of the FLINT library. These timings are also available in the form of plots with Figures 1 and 2 where radix-2 logscales are used on both axes.

Fig. 1: Multi-point timings (using radix-2 log-scales on both axes): CUMODP vs FLINT.

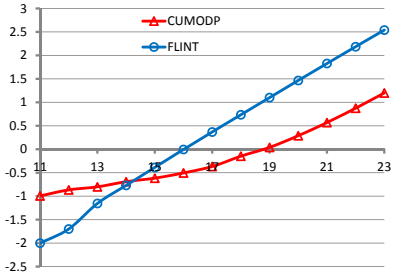
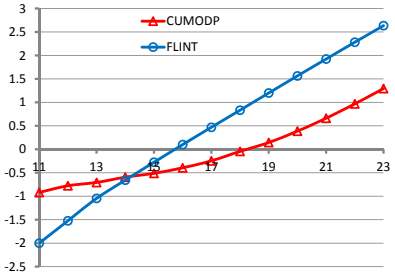


Fig. 2: Interpolation timings (using radix-2 log-scales on both axes): CUMODP vs FLINT.



We found that our implementation does not perform well until degree 2^{15}. In degree 2^{23}, we achieve a 21 times speedup factor w.r.t. FLINT, which is a satisfactory result. Nevertheless, we believe that by improving our multiplication routine for polynomials of degrees 2^9 to 2^{13}, we would have better performance in both polynomial evaluation and interpolation in these middle ranges.

8 Conclusion

We discussed fast multi-point evaluation and interpolation of univariate polynomials over a finite field on GPU architectures. We have combined algorithmic techniques like subproduct trees, subinverse trees, plain polynomial arithmetic, FFT-based polynomial arithmetic. Up to our knowledge, this is the first report on a parallel implementation of subproduct tree techniques. The source code of our algorithms is freely available in CUMODP-Library website `http://cumodp.org/`.

The experimental results are promising. Room for improvement, however, still exists, in particular for efficiently multiplying polynomials in the range of degrees from 2^9 to 2^{13}. Filling this gap is work in progress.

References

[1] D. J. Bernstein. Fast multiplication and its applications, 2008. In Algorithmic number theory: lattices, number fields, curves and cryptography, edited by Joe Buhler, Peter Stevenhagen.

[2] A. Bostan and É. Schost. Polynomial evaluation and interpolation on special sets of points. *J. Complexity*, 21(4):420–446, 2005.

[3] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in gf(2)[x]. In *Proceedings of the 8th international conference on*

Table 3: Multi-point evaluation and interpolation timings (in sec.) with input size 2^k: CUMODP vs FLINT.

	Evaluation			Interpolation		
k	GPU (s)	FLINT (s)	Ratio	GPU (s)	FLINT (s)	Ratio
11	0.1012	0.01	0.0987	0.1202	0.01	0.0831
12	0.1361	0.02	0.1468	0.1671	0.03	0.1794
13	0.1580	0.07	0.4429	0.1963	0.09	0.4584
14	0.2034	0.17	0.8354	0.2548	0.22	0.8631
15	0.2415	0.41	1.6971	0.3073	0.53	1.7242
16	0.3126	0.99	3.1666	0.4026	1.26	3.1294
17	0.4285	2.33	5.4375	0.5677	2.94	5.1780
18	0.7106	5.43	7.6404	0.9034	6.81	7.5379
19	1.0936	12.63	11.5484	1.3931	15.85	11.3768
20	1.9412	29.2	15.0420	2.4363	36.61	15.0268
21	3.6927	67.18	18.1923	4.5965	83.98	18.2702
22	7.4855	153.07	20.4486	9.2940	191.32	20.5851
23	15.796	346.44	21.9321	19.6923	432.13	21.9441

Algorithmic number theory, ANTS-VIII'08, pages 153–166, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, ISSAC '05, pages 108–115, New York, NY, USA, 2005. ACM.

[5] J. Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[6] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm i. *Appl. Algebra Eng., Commun. Comput.*, 14(6):415–438, 2004.

[7] S. A. Haque and M. Moreno Maza. Plain polynomial arithmetic on GPU. In *J. of Physics: Conf. Series*, volume 385, IOP Publishing, 2012.

[8] S. A. Haque, M. Moreno Maza, and N. Xie. A many-core machine model for designing algorithms with minimum parallelism overheads. *CoRR*, abs/1402.0264, 2014.

[9] W. B. Hart. Fast library for number theory: An introduction. In K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, editors, *Third International Congress on Mathematical Software, Proceedings*, volume 6327 of *Lecture Notes in Computer Science*, pages 88–91. Springer, 2010.

[10] X. Li, M. Moreno Maza, R. Rasheed, and Éric Schost. The Modpn library: Bringing fast polynomial arithmetic into Maple. *J. Symb. Comput.*, 46(7):841–858, July 2011.

[11] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. *Int. J. Found. Comput. Sci.*, 22(5):1035–1055, 2011.

[12] P. L. Montgomery. *An FFT Extension of the Elliptic Curve Method of Factorization.* PhD thesis, University of California Los Angeles, USA, 1992.

[13] M. Moreno Maza and W. Pan. Fast polynomial arithmetic on a GPU. *J. of Physics: Conference Series*, vol. 256, 2010.

[14] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a GPU. *J. of Physics: Conference Series*, vol. 341, 2011.

[15] H. Murao and T. Fujise. Towards an efficient implementation of a fast algorithm for multipoint polynomial evaluation and its parallel processing. In *Proc. of PASCO*, pages 24–30. ACM, 1997.

[16] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[17] S. Tanaka, T. Chou, B.-Y. Yang, C.-M. Cheng, and K. Sakurai. Efficient parallel evaluation of multivariate quadratic polynomials on GPUs. In *WISA*, pages 28–42, 2012.

[18] J. Verschelde and G. Yoffe. Evaluating polynomials in several variables and their derivatives on a GPU computing processor. In *Proc. of the 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1397–1405, 2012. IEEE Computer Society.