ICICLE v2: Polynomial API for Coding ZK Provers to Run on Specialized Hardware

Karthik Inbasekar, Yuval Shekel, Michael Asa

Abstract—Polynomials play a central role in cryptography. In the context of Zero Knowledge Proofs (ZKPs), protocols can be exclusively expressed using polynomials, making them a powerful abstraction tool, as demonstrated in most ZK research papers. Our first contribution is a high-level framework that enables practitioners to implement ZKPs in a more natural way, based solely on polynomial primitives.

ZK provers are considered computationally intensive algorithms with a high degree of parallelization. These algorithms benefit significantly from hardware acceleration, and deployed ZK systems typically include specialized hardware to optimize the performance of the prover code. Our second contribution is leveraging our polynomial API to abstract away low-level hardware primitives and automate their memory management. This device-agnostic design allows ZK engineers to prototype and build solutions while taking advantage of the performance gains offered by specialized hardware, such as GPUs and FPGAs, without needing to understand the hardware implementation details.

Finally, our polynomial API is integrated into version 2 of the ICICLE library [1] and is running in production. This paper also serves as a comprehensive documentation for the ICICLE v2 polynomial API.

1 Introduction

Zero Knowledge Proofs (ZKP) are cryptographic protocols that enable one party (prover) to prove to another party (verifier) that a given computation was executed correctly without revealing any information other than the statement to be proven. For resource intensive computations, verification by re-execution is simply not feasible. ZKPs possess the succinctness property as a consequence of which verification algorithms for ZKPs are exponentially faster than re-executing the original program. The succinctness and zero Knowledge properties of a ZKP have truly ground breaking consequences for applications such as anonymous authorization and payments, trust-less compute in the cloud, privacy and scaling in blockchains, verifiable machine learning/inference, etc.

In order to generate a ZKP, a program is first compiled into an intermediate representation of the form $C(x, w) = y$ where x, y are public, and w is a private witness. In general, there are two different front-end paradigms for converting a program into the intermediate representation. These are known as the ASIC and CPU approaches [1]. In the ASIC approach, a program is compiled into an equivalent hard coded arithmetic circuit consisting of gate and wire constraints. This is suited for structured computations with repetitive and deterministic program structure. Examples of these are often in blockchain applications [2]. In the CPU paradigm, the front end uses compilers to convert programs written in high level languages such as Rust/Go/C++ into assembly code for a given ISA (Instruction Set Architecture) such as RISC-V [2] or WebAssembly [3]. Thus proving a program execution is equivalent to proving the execution of the ISA corresponding to the program. This approach is more suited for programs where the logic is designed by the user. Examples of these are verifiable computation in Virtual Machine frameworks[3]. In either case, based on a prover supplied private input, the front end generates an execution trace with all intermediate values in a read only format.

The backend of a ZKP compiles the algebraic constraint relations of the circuit or ISA and memory access in VM into polynomial identities and checks that witness values in the execution trace obey these identities. It then uses a combination of either

karthik@ingonyama.com
yuval.shekel@ingonyama.com
miki.asa@ingonyama.com

1. See §A table 3 for a comparison

2. See §A table 4 for a list of commonly used frameworks in the ASIC approach with frontends and their supported backends

3. See §A table 5 for a list of popular zkVM frameworks in the CPU approach with frontends and their supported backends

- Polynomial IOP (Interactive Oracle Proofs)+ PCS (Polynomial commitment schemes)[4]
- Linear PCP (Probabilistically Checkable Proofs) + pairing based cryptography

to generate a cryptographic proof (ZKP) for circuit satisfiability (CSAT) or valid state transitions of the computation. The completeness and soundness property of a ZKP ensures that a proof generated by an honest prover will always pass the verification, whereas the probability of success for a malicious prover is negligible. Some of the backends used in production are [4], [5], [6], [7], [8], [9], [10] [5]. The main focus of this paper is to introduce new device-agnostic tools for efficient backend computation, tailored to the developers.

The backend computation that generates a ZKP is orders of magnitude more compute and memory intensive than simply re-executing the original computation. Hardware acceleration that utilizes inherent symmetries and structures in the cryptographic primitives is crucial for scalable applications. In general, the bulk of the computational burden in a PIOP comes from PCS and polynomial constraint evaluations. Fortunately, most of the protocols and sub-protocols that underlie PIOPs/PCPs are decomposable into algorithms that have a SIMD (Single Instruction Multiple Data) structure, suited for SIMD-compatible HW such as GPU. Here we refer to algorithms with parallelizable structures as hardware primitives. In table 1, we have summarized commonly used cryptographic commitment schemes and their underlying hardware primitives supported by the ICICLE library from Ingonyama [1]. ICICLE is a fully featured ZK hardware acceleration library, consists of CUDA kernels, with c++, rust, and go wrappers for all the non-lattice hardware primitives listed in table 1. ICICLE is fully open source, and is available with MIT license. ICICLE has been integrated into several ZK based products, such as [11], [12].

In this paper we introduce a new key feature of the ICICLE library, namely the Polynomial API. Cryptographic protocols in academic papers express PIOPs and PCPs naturally using polynomials. However, implementing them in hardware often forces researchers and developers to delve deep into implementations of hardware primitives that underly cryptographic protocols, both for prototyping and efficiency.

4. AHP (Algebraic Holographic Proofs) are special cases of PIOP are also included in this class.
5. See table 4 and 5 for a quick summary.

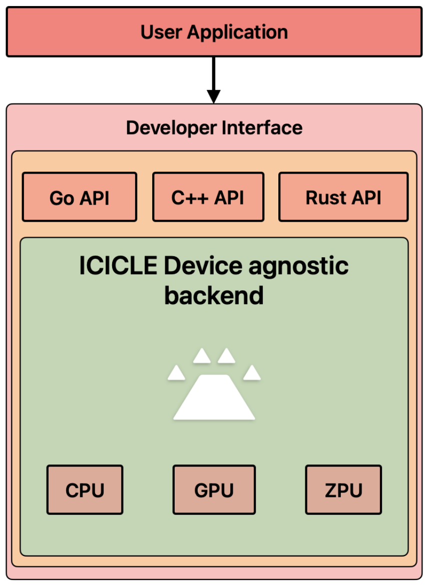


Fig. 1. ICICLE Polynomial API device agnostic developer interface. ZPU stands for the Zero Knowledge Processing Unit [21]

The ICICLE Polynomial API provides high level functionality for polynomial arithmetic, and abstracts away hardware complexities under the hood. The main motivations for the polynomial API are:

- To enable researchers to prototype cryptographic protocols in the polynomial language used to express PIOP/PCP, with the added benefit of prototyping in different hardware environments.
- To enable developers to efficiently implement cryptographic protocols and quickly build high performing applications, by abstracting away hardware complexities in a device agnostic manner (see fig 1).
- To enable end-to-end device implementation of ZKP backends, providing a easy way to overcome host-device data transfer bottlenecks without requiring the user to explicitly handling the memory management.

The general idea of expressing code using Polynomials as an API language has been pursued in ZKP in many frameworks such as Arkworks [22], Gnark [11], Polynomial Identity Language (PIL) [23], Plonky2/3 [7], [8] to name just a few. All of these frameworks are specific to CPUs. ICICLE, on the other hand, provides a complete framework for ZKP backends using the polynomial abstractions, with device agnostic hardware support for multiple devices. In Fully Homomorphic Encryption (FHE) the ecosystem already uses polynomial

TABLE 1
Hardware primitives for commonly used PCS (Polynomial Commitment Schemes). In the current version of ICICLE, vector ops include common linear algebra operations such as matrix multiplication and more generally tensor operations in future. FRI is an IOP that is also used as a commitment scheme

Cryptography	Scheme	Hardware primitives
Pairing based	univariate KZG [13]	MSM, NTT
	Zeropmorph (MLE)	vector ops, MSM
	KZG (MLE) [14]	vector ops, MSM
Discrete log	Pedersen commitment	MSM
	IPA [15]	MSM
	Brakedown [16]	Vector ops, MSM
	Hyrax [17]	Vector Ops, MSM
Collision resistant hashes	Merkle commitment [18]	hash functions, Merkle tree
	FRI [19]	hash functions, Merkle tree, NTT
Lattice based (Ring)	SIS hash [20]	Ring arithmetic, vector ops, Ring NTT

APIs in different forms including hardware support. The Hardware Abstraction Layer (HAL) of OpenFHE [24] supports multiple backends such as GPU/FPGA. The tfhe-rs library [25] uses polynomial API's with GPU support. In more recent works, Polynomial Instruction Based Compiler [26] and Homomorphic Encryption Intermediate Representation (HEIR) [27] also use polynomial API based framework. While FHE, and more broadly, lattice based cryptosystems [28], [29], [30] and their polynomial APIs, provide a good reference and inspiration, ZKPs programs, arithmetic, and hardware resources have different requirements. we keep the design of a unified framework, as well as the feasibility question, as a future work.

In fig 2 we illustrate the ease of use of the polynomial API for the quotient argument of Groth16 [4] (more details in §4). The quotient argument computation involves 3 INTTs and 3 NTTs on a coset, followed by Hadamard product, polynomial arithmetic and division by the vanishing polynomial (35). On the RHS of fig 2 we see an implementation in Gnark [11] using the low level API's for NTT/INTT and division algorithms. The code is about 50 lines which involves also explicit memory allocation and memory management on the device (GPU). On the LHS, below the quotient polynomial dataflow, the polynomial API essentially achieves the same in just a single line of code! Note that the API language literally follows the quotient argument verbatim, while efficiently managing the entire data flow, NTTs/INTTs and memory allocation under the hood!

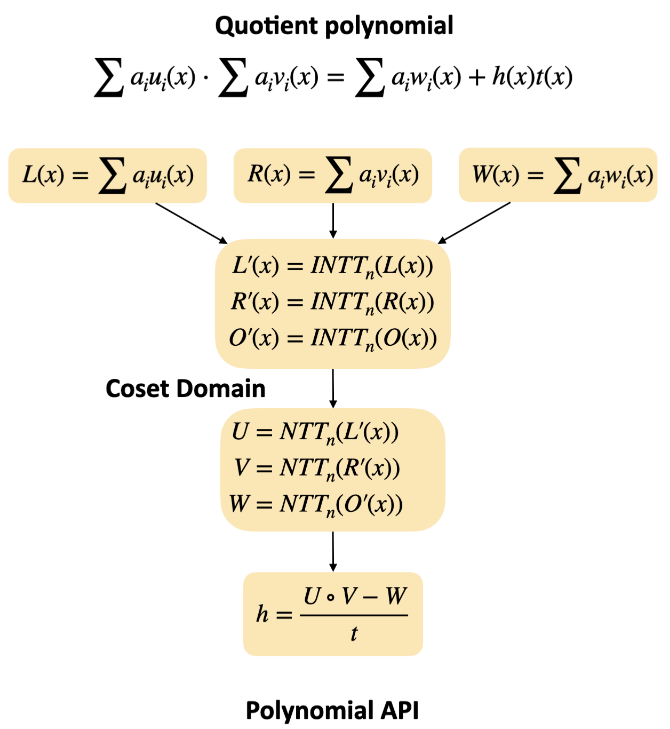
While much of the compute algorithms are parallel, the dataflow between host and device can still downgrade performance significantly. The Polynomial API achieves efficient data handling through the use of memory views (an integrity pointer to data, stored on device memory) that provide direct **read-only access** to the polynomial's internal state without the need to explicitly copy data. Using memory views, a user can read data from memory and perform operations such as cosetNTT, commitments (MSM), Merkle trees, hashes etc, without moving/copying the data between device and host. This feature is key to realizing end-to-end PIOP/PCP in specialized hardware. Last but not the least, the API is device agnostic - it allows efficient integration of different backends without changing the code base.

The paper is organized as follows. In §2 we begin with a quick overview of the ICICLE library (Version 2.x). We discuss the supported curves and operations. In §3 we introduce the features of the polynomial API. We cover some relevant background on polynomials in finite field in §C. For the purposes of this paper, we discuss an explicit end-to-end prover example using groth16 [4] §4 in C++. We summarize in §**??**.

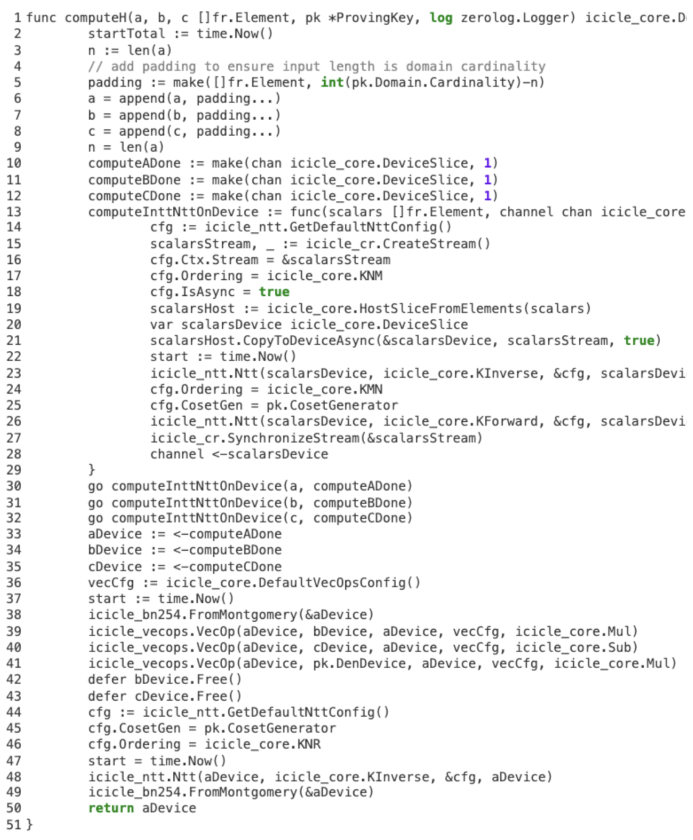
2 ICICLE LIBRARY OVERVIEW

ICICLE [1] is a fully featured accelerated cryptography library for ZKP. In the current iteration, ICICLE supports GPU acceleration by implementing in native CUDA code all the hardware primitives from table 1 and their underlying base/scalar field operations. The "stacked tile" structure of the ICICLE library is summarized in fig 3. At the core level are the CUDA kernels for all hardware primitives from table 1.

- All yellow tiles represent computations involving modular arithmetic in the scalar field $\mathbb{F}_r$, where r is the characteristic of the scalar field.



```
h = (u * v − w).divide_by_vanishing_polynomial(degree) ;
```



```
func computeH(a, b, c []fr.Element, pk *ProvingKey, log zerolog.Logger) icicle_core.D
        startTotal := time.Now()
        n := len(a)
        // add padding to ensure input length is domain cardinality
        padding := make([]fr.Element, int(pk.Domain.Cardinality)-n)
        a = append(a, padding...)
        b = append(b, padding...)
        c = append(c, padding...)
        n = len(a)
        computeADone := make(chan icicle_core.DeviceSlice, 1)
        computeBDone := make(chan icicle_core.DeviceSlice, 1)
        computeCDone := make(chan icicle_core.DeviceSlice, 1)
        computeInttNttOnDevice := func(scalars []fr.Element, channel chan icicle_core
                cfg := icicle_ntt.GetDefaultNttConfig()
                scalarsStream, _ := icicle_cr.CreateStream()
                cfg.Ctx.Stream = &scalarsStream
                cfg.Ordering = icicle_core.KNM
                cfg.IsAsync = true
                scalarsHost := icicle_core.HostSliceFromElements(scalars)
                var scalarsDevice icicle_core.DeviceSlice
                scalarsHost.CopyToDeviceAsync(&scalarsDevice, scalarsStream, true)
                start := time.Now()
                icicle_ntt.Ntt(scalarsDevice, icicle_core.KInverse, &cfg, scalarsDevi
                cfg.Ordering = icicle_core.KMN
                cfg.CosetGen = pk.CosetGenerator
                icicle_ntt.Ntt(scalarsDevice, icicle_core.KForward, &cfg, scalarsDevi
                icicle_cr.SynchronizeStream(&scalarsStream)
                channel <-scalarsDevice
        }
        go computeInttNttOnDevice(a, computeADone)
        go computeInttNttOnDevice(b, computeBDone)
        go computeInttNttOnDevice(c, computeCDone)
        aDevice := <-computeADone
        bDevice := <-computeBDone
        cDevice := <-computeCDone
        vecCfg := icicle_core.DefaultVecOpsConfig()
        start := time.Now()
        icicle_bn254.FromMontgomery(&aDevice)
        icicle_vecops.VecOp(aDevice, bDevice, aDevice, vecCfg, icicle_core.Mul)
        icicle_vecops.VecOp(aDevice, cDevice, aDevice, vecCfg, icicle_core.Sub)
        icicle_vecops.VecOp(aDevice, pk.DenDevice, aDevice, vecCfg, icicle_core.Mul)
        defer bDevice.Free()
        defer cDevice.Free()
        cfg := icicle_ntt.GetDefaultNttConfig()
        cfg.CosetGen = pk.CosetGenerator
        cfg.Ordering = icicle_core.KNR
        start = time.Now()
        icicle_ntt.Ntt(aDevice, icicle_core.KInverse, &cfg, aDevice)
        icicle_bn254.FromMontgomery(&aDevice)
        return aDevice
}
```

Fig. 2. LHS: Polynomial API expresses quotient argument and division by vanishing polynomial (23) in a single line of code, verbatim from the paper [4], while managing all the NTT's/INTT's, vector operations and memory management under the hood. RHS: The same computation using low level NTT/iNTT APIs is a longer code, and requires explicit memory management.

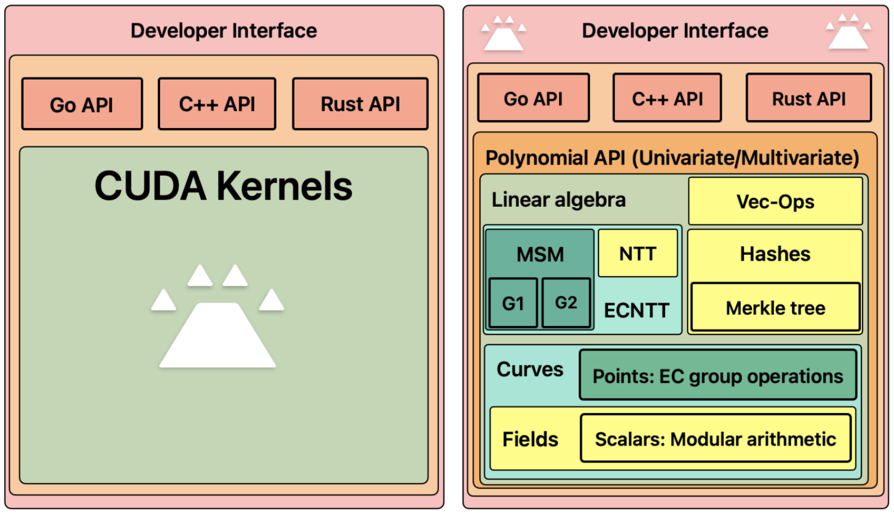


Fig. 3. ICICLE structure: ICICLE backend for hardware primitives are at the core of ICICLE library. Developers can directly access the GPU accelerated functionality via the APIs in C++/Rust/Go without having to deal with CUDA code.

- All dark green tiles represent computations in group operations in $\mathbb{G}_1, \mathbb{G}_2$ and base field arithmetic $\mathbb{F}_q$, where q is the characteristic of the base field.
- The teal green tile "curves" is an umbrella definition for all finite field operations i.e in both $\mathbb{F}_r$ and $\mathbb{F}_q$. ICICLE is a static compiled library compiled per curve, or field. The supported curves and primitives are summarized in table 2.

- Built on top of it is the Linear algebra tile (for classification purposes) , which consists of the MSM, ECNTT kernels in $\mathbb{F}_q$ and Vec-Ops kernels in $\mathbb{F}_r$. The ECNTT (Elliptic Curve Number Theoretic Transform) does NTT on a vector of group elements and can be used for instance to transform a Structured Reference String (SRS) from monomial to Lagrange basis.
- The hashes and Merkle tree tiles involve both linear and non-linear operations but only in $\mathbb{F}_r$. Currently ICICLE v2.x supports the Keccak and finite field friendly Poseidon and Poseidon2 hash. The supported modes are sponge mode, Merkle tree based hashing and commitments.
- The polynomial API tile is an over arching layer which can access any of the CUDA kernels related to MSM, NTT, Vec-ops, hashes and Merkle tree. Moreover, the API is open to extension and is not limited to a predefined set of operations. The API is available in C++ with shallow wrappers to Rust and Go.

ICICLE is a statically compiled library, clone the ICICLE library [1] and compile for a specific curve or field (see appendix §B for a quick starter).

3 Polynomial API

The Polynomial API is a framework for polynomial operations of a given datatype.[6] For the purpose of presentation we limit the discussion of the API in C++. The API is accessible via wrapper functionalities in rust and go [7]. In C++ API, the Polynomial class[8] defines a polynomial with the following template

```
template <typename Coeff, typename
↪   Domain = Coeff, typename Image =
↪   Coeff>
class Polynomial {
    // Polynomial class definition
}
```

where

- **Coeff**: Coefficients
- **Domain**: Specifies the type for the input values over which the polynomial is evaluated.
- **Image**: Defines the type of the output values of the polynomial. By default, the image is of the same type as that of the coefficients

The allowed data type names are summarized below. These cover the scalars and points described in the ICICLE overview (see fig 3)

- `scalar_t` : $\mathbb{F}_r$ (default)
- `point_field_t` : $\mathbb{F}_q$
- `affine_t` : curve points $(x, y) \in \mathbb{G}_1$, where $x, y \in \mathbb{F}_q$.
- `g2_affine_t` : Affine curve points $(x, y) \in \mathbb{G}_2$, where $x, y \in \mathbb{F}_q$
- `projective_t` : Projective curve points $(X, Y, Z) \in \mathbb{G}_1$, where $X, Y, Z \in \mathbb{F}_q$
- `g2_projective_t` : Projective curve points $(X, Y, Z) \in \mathbb{G}_2$, where $X, Y, Z \in \mathbb{F}_q$

Each data type supports operator overriding arithmetic relevant to its data type. i.e modular arithmetic in $\mathbb{F}_r$, modular arithmetic in $\mathbb{F}_q$, EC group additions and scalar multiplications in all the curve types. Creation of random field elements, unity, zero etc can be done using

```
//create random field element in 𝔽r
auto scalar = scalar_t::rand_host();
//create field element from specific
↪  data
auto scalar = scalar_t::from(u32);
// Projective generators of 𝔾1
auto point = projective_t::generator();
```

and conversions for `projective_t` to `affine_t` and vice versa, are all accessible within the polynomial API. The templated structure is very powerful and can accommodate different data types for coefficients, the domain, and images.

We summarize the main features of the polynomial API below. We have included relevant theory sections in the appendix §C for quick reference.

- **Construction**: §3.2 Create polynomials in coefficients (25) or evaluations form (26).
- **Arithmetic Operations**: §3.3 Perform addition, subtraction, multiplication, and division as discussed in §C.2.
- **Evaluation**: §3.4 Directly evaluate polynomials at specific points or across a domain (24).
- **Manipulation**: §3.5 Polynomial degree, §3.6 Slicing polynomials, adding or subtracting monomials inplace.

6. See §C for a quick review of Polynomials in Finite Fields.
7. https://dev.ingonyama.com/icicle/rust-bindings,https://dev.ingonyama.com/icicle/golang-bindings, Note that only `scalar_t` : $\mathbb{F}_r$ is exposed in rust and go wrappers
8. See https://github.com/ingonyama-zk/icicle/blob/main/icicle/include/polynomials/polynomials.h

TABLE 2
Curve/field specific hardware primitives currently supported by ICICLE

Curves/ Primitives	BN254	BLS12-377	BLS12-381	BW6-761	Grumpkin	Baby bear	Stark252
MSM $\mathbb{G}_1$	✓	✓	✓	✓	✓	–	–
MSM $\mathbb{G}_2$	✓	✓	✓	✓	×	–	–
NTT	✓	✓	✓	✓	×	✓	✓
ECNTT	✓	✓	✓	✓	×	–	–
VecOps	✓	✓	✓	✓	✓	✓	✓
Merkle Tree (Poseidon hash)	✓	✓	✓	✓	✓	✓	✓
Extension Field	–	–	–	–	–	✓	–

- **Memory Access**: §3.7 Access internal states of the polynomial and copy data from device to host.
- **Memory views**: §3.8 Device-memory views of polynomials that access polynomial interal state directly, without making copies or moving data. Read device memory and perform external operations such as MSM, Merkle tree, hashes, using the respective API. Manage memory views through integrity pointers to safeguard against stale or non-existent data.

In polynomial construction, arithmetic, evaluation and manipulations, the relevant computation using NTT/INTT and storage (on device/off device) is abstracted away. This enables adaptability to various backends in general. Currently the API supports a CUDA backend, but the design allows to load/switch backends at runtime as per the user's choice. This capability allows users to perform polynomial operations without the need to tailor their code to specific hardware and generically work with the Polynomial API.

3.1 Instantiation and usage

Instantiation is straightforward, and we provide a starter template to quickly get started. As always the ICICLE library should be statically compiled first and the relevant files linked (see appendix §B in the `Cmake` file. Initialization with an appropriate factory per linked curve/field is required to configure the computational context and backend. For the rest of the paper we will assume that the linked curve is `bn254`.

```
//include relevant header files and
↪  cuda header files
#include "polynomials/polynomials.h"
#include "polynomials/cuda_backend/pol⌋
↪  ynomial_cuda_backend.cuh"
#include "ntt/ntt.cuh"
#include "api/bn254.h"

//access data_types using namespace
using namespace bn254;
using namespace polynomials;

  //define polynomial data type:
↪  defines a polynomial with data
↪  type 𝔽_r
typedef Polynomial<scalar_t>
↪  Polynomial_t;
```

It is important to define the domain of the computation (24) before instantiating the polynomials. The domain is handled by the NTT config [9], and the roots of unity are stored on device automatically.

```
//config domain
const int MAX_NTT_LOG_SIZE = 24;
auto ntt_config = ntt::default_ntt_con⌋
↪  fig<scalar_t>();
const scalar_t basic_root =
↪  scalar_t::omega(MAX_NTT_LOG_SIZE);
ntt::init_domain(basic_root,
↪  ntt_config.ctx);
// Initialize with a CUDA backend
Polynomial_t::initialize(std::make_sha⌋
↪  red<CUDAPolynomialFactory<>>());
```

It is generally recommended to initialize the final domain if possible, since at the time of initialization the roots of unity for the domain are stored in the device automatically. This enables faster computation and efficient memory management. The abstract factory is a template for creating polynomial

9. https://github.com/ingonyama-zk/icicle/blob/main/icicle/include/ntt/ntt.cuh. The config has an option to modify coset generators, to do coset NTT, we will see an example later.

contexts and backends. [10] In this paper, we will use the templater only with `scalar_t` type as it is the most common use case.

3.2 Construction

Polynomials can be constructed from coefficients in $\mathbb{F}_r$ or from evaluations on roots-of-unity domains $H_n \subset F_r^{\times}$. This method, constructs an array of coefficients/evaluations on roots of unity domain, of a given size on the device, and returns a pointer to the newly created polynomial instance.

```
from_coefficients(const Coeff*
 ↪  coefficients, uint64_t
 ↪  nof_coefficients);
from_rou_evaluations(const Image*
 ↪  evaluations, uint64_t
 ↪  nof_evaluations);
```

Note that the domain and polynomial factory are instantiated as discussed in §3.1. Below is a simple example for creating a random polynomial of arbitrary size

```
// Defines a polynomial instance based
 ↪  on the scalar type from the field
 ↪  configuration
  typedef Polynomial<scalar_t>
   ↪  Polynomial_t;
// Construction of a random polynomial
static Polynomial_t
 ↪  randomize_polynomial(uint32_t size)
{
  auto coeff = std::make_unique<scalar⌟
   ↪  _t[]>(size);
  for (int i = 0; i < size; i++)
    elements[i] =
     ↪  scalar_t::rand_host();
  //for coefficients form use
  return Polynomial_t::from_coefficien⌟
   ↪  ts(elements.get(),
   ↪  size);
  //for evaluations form use
  return Polynomial_t::from_rou_evalua⌟
   ↪  tions(elements.get(),
   ↪  size)
}
```

Another way to create polynomials is to clone existing instances. The clone function takes the pointer to the given instance, creates a copy of the polynomial, and returns a pointer to the copy. Note that, the clone and the original do not share memory. This is not recommended unless there are branches in the computation, i.e a given polynomial has two different evolution routes. We provide more efficient methods in §3.7 and §3.8

```
auto f1 =
 ↪  randomize_polynomial(uint32_t
 ↪  size);
auto f1_cloned = p.clone(); //
 ↪  f1_cloned and f do not share memory
```

removing polynomial instances from memory if needed can be achieved using the `.delete()` method.

3.3 Arithmetic

All the usual polynomial arithmetic operations (see §C.2) are possible within any instantiated data type. In the regular arithmetic mode, the functions take a pointer to the two polynomials to add and outputs a pointer to the result of the addition. The inplace addition method for addition/subtraction can be used as usual with the `+=` operator. In this case. the first pointer, points to the result.

```
// Addition
Polynomial operator+(const Polynomial&
 ↪  rhs) const;
// inplace addition
Polynomial& operator+=(const
 ↪  Polynomial& rhs);

// Subtraction
Polynomial operator-(const Polynomial&
 ↪  rhs) const;

// Multiplication
Polynomial operator*(const Polynomial&
 ↪  rhs) const;

// scalar multiplication
Polynomial operator*(const Domain&
 ↪  scalar) const;

// Division A(x) = B(x)Q(x) + R(x)
std::pair<Polynomial, Polynomial>
 ↪  divide(const Polynomial& rhs)
 ↪  const; // returns (Q(x), R(x))

// returns quotient Q(x) only
Polynomial operator/(const Polynomial&
 ↪  rhs) const;
```

10. https://github.com/ingonyama-zk/icicle/blob/main/icicle/include/polynomials/polynomial_abstract_factory.h

```
// returns remainder R(x) only
Polynomial operator%(const Polynomial&
 ↪  rhs) const;
```

In the example below, we create a random polynomials $f_1(x), f_2(x)$ of degree $N-1$ and verify the identity

$$(f_1(x)+f_2(x))^2+(f_1(x)-f_2(x))^2 = 2\cdot(f_1(x)^2+f_2(x)^2) \quad (1)$$

```
//define random poly in coeff form or
 ↪  eval form
auto f1 = randomize_polynomial(N);
auto f2 = randomize_polynomial(N);

//deg 2N constraints (f1+f2)^2 +
 ↪  (f1-f2)^2 = 2 (f1^2+ f_2^2)
auto L1 = (f1+f2)*(f1+f2) +
 ↪  (f1-f2)*(f1-f2);
auto R1 = scalar_t::from(2) * (f1*f1 +
 ↪  f2*f2);

//some assertion method
assert_eq(L1==R1);
```

Note that the code is verbatim (1). Note also that the initial domain in this example §3.1 has to be atleast `MAX_NTT_LOG_SIZE = 2 * N`.

In ZKPs polynomial identity checking is implemented in PIOP/PCP with the quotient argument (see §C.3). This involves evaluating the constraints followed by division with the vanishing polynomial (23) in an appropriate domain

```
// division by the vanishing
Polynomial divide_by_vanishing_polynom⌋
 ↪  ial(uint64_t degree)
 ↪  const;
```

Below the quotient argument of Groth16 [4] discussed in (34) is a single line code.

```
auto H = (A*B-C).divide_by_vanishing_p⌋
 ↪  olynomial(N);
```

Usually this computation is done in a coset domain. An explicit coset generator can be specified as follows and coset domain can be configured

```
//config base domain
const int MAX_NTT_LOG_SIZE = 24;
auto ntt_config = ntt::default_ntt_con⌋
 ↪  fig<scalar_t>();
const scalar_t basic_root =
 ↪  scalar_t::omega(MAX_NTT_LOG_SIZE);

  //Instantiate A,B,C polynomials such
 ↪  that A(x)·B(x)≡C(x) in base domain

//modify coset generator
ntt_config.coset_gen = ntt::get_root_o⌋
 ↪  f_unity<scalar_t>(size *
 ↪  2);

//config coset domain
ntt::init_domain(basic_root,
 ↪  ntt_config.ctx);

//divide by vanishing poly in coset
 ↪  domain
auto H = (A*B-C).divide_by_vanishing_p⌋
 ↪  olynomial(N);
```

Note that the all the cosetNTTs/cosetINTT (35) and memory allocation are automatically taken care of. The data in the results of the arithmetic remains in the device memory and can be accessed using views §3.8 or sent to the host §3.7.

3.4 Evaluation

Polynomials can be evaluated at arbitrary $c \in \mathbb{F}_r$, or in a roots of unity domain H_n or a coset domain $H_{n,\eta}$,

```
//evaluate on random point
Image operator()(const Domain& x)
 ↪  const; //
void evaluate(const Domain* x, Image*
 ↪  evals /*OUT*/) const;

//evaluate on a given set of scalars
 ↪  (not necessarily Roots of
 ↪  unity/coset). Note that memory is
 ↪  allocated on call.
void evaluate_on_domain(Domain*
 ↪  domain, uint64_t size, Image*
 ↪  evals /*OUT*/) const;

//evaluate on a given set of domains
 ↪  (equivalent to NTT)
//Roots of unity/coset domain already
 ↪  on device (faster)
void evaluate_on_rou_domain(uint64_t
 ↪  domain_log_size, Image* evals
 ↪  /*OUT*/) const;
```

Some examples of the above are

```
//define random poly in coeff form
auto f = randomize_polynomial(N);

//evaluate on random point
scalar_t x = scalar_t::rand_host();
auto f_x = f(x);

// evaluate f on a arbitrary set of
↪ points
auto domain =
↪ std::make_unique<scalar_t[]>(size);
for (int i = 0; i < N; ++i) {
    domain[i] = scalar_t::rand_host();
}
auto evaluations =
↪ std::make_unique<scalar_t[]>(N);
f.evaluate_on_domain(domain, N,
↪ evaluations.get());

// evaluate f(x) on roots of unity
↪ domain (equivalent to NTT)
uint64_t domain_log_size = N;
auto evaluations_rou_domain =
↪ std::make_unique<scalar_t[]>(1 <<
↪ domain_log_size);
f.evaluate_on_rou_domain(domain_log_si⌋
↪ ze,
↪ evaluations_rou_domain);
```

3.5 Manipulations

Given a polynomial, the monomial operations allow to access specific terms in the coefficient form and perform additions/subtractions, the `monomial_coeff` represents the value, and `monomial` represents the location in the array (degree of the term in the polynomial).

```
// Monomial operations
Polynomial& add_monomial_inplace(Coeff
↪ monomial_coeff, uint64_t monomial);
Polynomial& sub_monomial_inplace(Coeff
↪ monomial_coeff, uint64_t monomial);
```

For example

```
auto f = randomize_polynomial(N);
//add a coefficient to degree zero term
 f.add_monomial_in_place(scalar_t::fro⌋
↪ m(5)); //
↪ $f(x)+=5$
//add a coefficient to degree 8 term
 f.sub_monomial_in_place(scalar_t::fro⌋
↪ m(3), 8); //
↪ $f(x)-=3x^8$
```

The degree of a polynomial in coefficients form is the highest power of the variable with a non zero coefficient (see §C.1). The `degree()` function in the API returns the degree of the polynomial in coefficient form, corresponding to the highest exponent with a non-zero coefficient.

```
auto f = randomize_polynomial(N);
//outputs a poly of degree N-1
auto degree_of_f = f.degree();
```

Note that the constant polynomial has degree zero by definition and the zero polynomial has degree -1 in our convention.

3.6 Slicing - create copies on device

In polynomial operations, it is common to extract the odd and even indexed/degree terms. Or in general to access arbitrary slices starting from a given offset (starting index),a given stride length (interval between elements in a slice) and number of elements in a slice

```
// Slicing arbitary coefficients.
Polynomial slice(uint64_t offset,
↪ uint64_t stride, uint64_t size);
//slicing to even or odd components
Polynomial even();
Polynomial odd();
```

Below is a simple example of the folding with randomness, which appears in several PIOP arguments such as FRI. [9]

```
// folding odd and even components
↪ (powers) of a poly with random
↪ value
// $fold(F(x)) = F_e(x^2) + xF_o(x^2)$
auto x = rand_host();
auto even = f.even();
auto odd = f.odd();
auto fold_poly = even + odd * x;
```

An example using the strides and size arguments to extract arbitrary slices of polynomials (in device).

```
const scalar_t coeffs[4] = {one, two,
↪ three, four};
//f(x) = 1 + 2x + 3x^2 + 4x^3
auto f = Polynomial_t::from_coefficien⌋
↪ ts(coeffs,
↪ 4);
// extract slice 1 + 4x^3
auto f_slice = f.slice(0 /*=offset*/,
↪ 3 /*= stride*/, 2 /*/= size*/);
```

3.7 Memory access - move data between host/device

Since the polynomial is instantiated on the device, the data resides on the device memory. In several situations, such as memory bottlenecks, and concurrent CPU operations such as preprocessing, or to perform any device unsupported computations, access to the polynomial's internal state can be vital. The data copy function copies the polynomial coefficients to either host or device allocated memory.

```
// copy single coefficient device -->
↪ host
Coeff get_coeff(uint64_t idx) const;
// copy a range of coefficients device
↪ --> host
uint64_t copy_coeffs(Coeff* coeffs,
↪ uint64_t start_idx, uint64_t
↪ end_idx) const;
```

As a simple example copy of to host, we can copy the data from a specific slice to host

```
const scalar_t coeffs[4] = {one, two,
↪ three, four};
//f= 1+2x+3x^2+4x^3
auto f = Polynomial_t::from_coefficien⌋
↪ ts(coeffs,
↪ 4);
auto even = f.even();
//copy 1+3x^2 to host
const auto slice_nof_coeffs =
↪ f_slice.copy_coeffs(even, 0, 1);
```

In general we recommend using memory views as much as possible §3.8 to avoid dreaded memory and data-transfer bottlenecks.

3.8 Memory Views - operate from data in device memory without copying

A powerful feature of the Polynomial API is efficient data handling through the use of memory views. These views provide direct read only access to the polynomial's internal state **without the need to copy data**. This feature is particularly useful for operations that require direct access to device memory, enhancing both performance and memory efficiency.

A memory view is a pointer to data stored in device memory. By providing a direct access pathway to the data, it eliminates the need for data duplication, thus conserving both time and system resources. This is especially beneficial in high-performance computing environments where data size and operation speed are critical factors. This is key to design efficient end-end ZKP provers on specialized hardware devices, to have high device occupancy, and to avoid the dreaded data flow bottlenecks between host-device and vice-versa.

A view of the polynomial data can be obtained by using the `get_coefficients_view()` functionality

```
// Obtain a view of the polynomial's
↪ coefficients
std::tuple<IntegrityPointer<Coeff>,
↪ uint64_t /*size*/, uint64_t
↪ /*device_id*/>
get_coefficients_view();
```

The pointer can be used by other data compatible ICICLE API to read only access data and perform operations. As an example, let us generate a random polynomial, read the data using views and do a cosetNTT on it

```
auto f = randomize_polynomial(size);
auto [d_coeff, size, device_id] =
↪ f.get_coefficients_view();
//compute coset evaluations
auto coset_evals =
↪ std::make_unique<scalar_t[]>(size);
auto ntt_config = ntt::default_ntt_con⌋
↪ fig<scalar_t>();

// using the device data directly as a
↪ view
ntt_config.are_inputs_on_device = true;
ntt_config.coset_gen = ntt::get_root_o⌋
↪ f_unity<scalar_t>(size *
↪ 2);

//use data from coeff_view using get
↪ method
```

```
ntt::ntt(d_coeff.get(), size,
↪   ntt::NTTDir::kForward, ntt_config,
↪   coset_evals.get());
```

`ntt_config.are_inputs_on_device = true` ensures that the NTT API **knows** to follow the pointer to the data accessed by the coefficients view method. This accesses the polynomial state directly and the NTT call `ntt::ntt(d_coeff.get(),...)` gets to read the data pointed to by the pointer for its use.

Below we provide a simple example for commitments using memory view [11]. In this example we check the simple identity (1) using commitments.

$$[(f_1(x)+f_2(x))^2+(f_1(x)-f_2(x))^2]_1 = [2\cdot(f_1(x)^2+f_2(x)^2)]_1 \quad (2)$$

First we use polynomial views to give the MSM API the location of the data and the API uses the `get()` to read the data and compute the commitment.

```
//choose size
int N = 1025;
//generate random group elements
↪   string of length 2N
auto SRS = generate_SRS(2*N);

//Allocate memory on device (points)
affine_t* points_d;
cudaMalloc(&points_d,
↪   sizeof(affine_t)* 2 * N);
// copy SRS to device
cudaMemcpy(points_d, SRS.get(),
↪   sizeof(affine_t)* 2 * N,
↪   cudaMemcpyHostToDevice);

//test commitment equality
  //[(f1(x)+f2(x))^2+(f1(x)-f2(x))^2]1 =
↪   [2(f1(x)^2+f2(x)^2)]1
//using polynomial views and commit

auto f1 = randomize_polynomial(N);
auto f2 = randomize_polynomial(N);
auto L1 = (f1+f2)*(f1+f2) +
↪   (f1-f2)*(f1-f2);
auto R1 = scalar_t::from(2) * (f1*f1 +
↪   f2*f2);

// extract coeff using coeff view
auto [viewL1, sizeL1, device_idL1] =
↪   L1.get_coefficients_view();
auto [viewR1, sizeR1, device_idR1] =
↪   R1.get_coefficients_view();
```

11. See examples section in C++ in [1]

```
msm::MSMConfig config =
↪   msm::default_msm_config();
config.are_points_on_device = true;
config.are_scalars_on_device = true;

//host vars (for result)
projective_t hL1{}, hR1{};

//straightforward msm bn254 api:
bn254_msm_cuda(viewL1.get(),points_d,N⌋
↪   ,config,&hL1);
bn254_msm_cuda(viewR1.get(),points_d,N⌋
↪   ,config,&hR1);
```

The `config.are_points_on_device=true` and `config.are_points_on_device=true` ensure that the msm API can access the data from the polynomial view.

Some times the data accessed by a certain view, can become modified due to in-place operations, and this invalidates the original view. Memory views are managed through an integrity pointer that monitors the validity of the memory it points to. It can detect if the memory has been modified or released, thereby preventing unsafe access to stale or non-existent data. This is done in practice using some simple validity check methods

```
// Checks if current pointer is still
↪   valid
bool isValid() const;

// Retrieves the raw pointer or
↪   nullptr if pointer is invalid
const T* get() const;

// Dereferences the pointer. Throws
↪   exception if the pointer is
↪   invalid.
const T& operator*() const;

//Provides access to the member of the
↪   pointed-to object.
Throws exception if the pointer is
↪   invalid.
const T* operator->() const;
```

Although in general one does not need to use this, in situations where there are combinations of in place operations and memory views, this feature can be useful to prevent errors.

3.9 Multiple-GPU Support

The Polynomial API also supports multiple GPU environments. Current active device is set by

```
cudaSetDevice(int deviceID);
```

All subsequent operations that allocate or deal with polynomial data will be performed on this device. Polynomial data are located on the current CUDA device at the time of genesis. Device context must be correctly set before initiating any operation that involves memory allocation, else an exception is thrown.

```
// Set the device before creating
↪  polynomials
cudaSetDevice(0);
Polynomial p1 = Polynomial::from_coeff⌋
↪  icients(coeffs,
↪  size);
cudaSetDevice(1);
Polynomial p2 = Polynomial::from_coeff⌋
↪  icients(coeffs,
↪  size);
// Throws an exception if p1 and p2
↪  are not on the same device
auto p3 = p1 + p2;
```

Note that access to degree of a polynomial or performing in-place modifications, can be executed regardless of the current device setting, since they do not involve creation of new polynomials.

4 C++ GROTH EXAMPLE

In this section, we give a quick walkthrough of the Groth16 prover computation. The full example can be found in the link in the footnote [12]. We will focus purely on the back end part of the computation although parts of the setup, R1CS to QAP can still be done in ICICLE, it is not yet possible to write circuits directly in ICICLE. These features may be supported in future.

The main goal of the Groth16 prover is to convince the verifier of the knowledge of coefficients $a \in \mathbb{F}_r$ that satisfy the R1CS system

$$(u.a) \circ (v.a) = (w.a) \tag{3}$$

12. https://github.com/ingonyama-zk/icicle/blob/main/icicle/tests/polynomial_test.cu

the a are input wire vectors consisting of public and witness elements $\in \mathbb{F}_r$. We will use the following notation to label the wire vectors (with $a_0 = 1$)

$$\{a_0 | \underbrace{a_1, a_2, \dots a_l}_{\text{Public } \in \mathbb{F}^l} | \underbrace{a_{l+1}, \dots a_m}_{\text{Witness } \in \mathbb{F}^{m-l}} \} \tag{4}$$

which is written in the QAP language

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) \equiv \sum_{i=0}^{m} a_i w_i(X) + h(X) \cdot t(X) \tag{5}$$

where the $deg(t(X)) = n$ and

$$deg(A_i(X)) = deg(B_i(X)) = deg(C_i(X)) = n-1 \tag{6}$$

At the end of the setup process the prover gets access to the following data

$$\begin{aligned}
QAP &= \{u_i(x), v_i(x), w_i(x), t(x)\} \ \forall i = 0, 1 \dots, m \\
CRS &= \Big\{([u_i(\tau)]_1, [v_i(\tau)]_1, [v_i(\tau)]_2, [w_i(\tau)]_1, \\
&\quad , [\alpha]_1, [\beta]_1, [\delta]_1, \left\{ \left[\frac{\tau^i t(\tau)}{\delta} \right]_1 \right\}_{i=0}^{n-2}, \\
&\quad \{[vk_i']_1\}_{i=0}^m, \{[pk_i']_1\}_{i=0}^m \\
&\quad , [\beta]_2, [\gamma]_2, [\delta]_2, \{[\tau^i]_2\}_{i=0}^{n-1} \Big\} \\
[pk_i']_1 &= \left\{ \left[\frac{\beta \cdot u_i(\tau) + \alpha \cdot v_i(\tau) + w_i(\tau)}{\delta} \right]_1 \right\}_{i=l+1}^{m} \\
[vk_i']_1 &= \left\{ \left[\frac{\beta \cdot u_i(\tau) + \alpha \cdot v_i(\tau) + w_i(\tau)}{\gamma} \right]_1 \right\}_{i=0}^{l}
\end{aligned} \tag{7}$$

where the evaluations at τ (the secret scalar/toxic waste) are encoded in group elements in $\mathbb{G}_1, \mathbb{G}_2$ and are protected by the discrete log property. The setup is generated for computing the quotient argument in Lagrange basis of polynomials. The prover algorithm is simply

1) Choose random $r, s \in \mathbb{F}_r$
2) Compute quotient (see (35))

$$h(x) = \frac{U(X) * V(x) - W(x)}{t(X)} \tag{8}$$

3) compute proof element $[A]_1$

$$[A]_1 = \sum_{i=0}^{m} a_i \cdot [u_i(\tau)]_1 + [\alpha]_1 + +r \cdot [\delta]_1 \tag{9}$$

4) compute proof element $[B]_2, [B]_1$

$$[B]_2 = \sum_{i=0}^{m} a_i \cdot [v_i(\tau)]_2 + [\beta]_2 + s \cdot [\delta]_2 \tag{10}$$

$$[B]_1 = \sum_{i=0}^{m} a_i \cdot [v_i(\tau)]_1 + [\beta]_1 + s \cdot [\delta]_1 \tag{11}$$

5) compute proof element $[C]_1$

$$[C]_1 = \sum_{i=l+1}^{m} a_i \cdot [pk_i']_1 + \sum_{i=0}^{n-2} h_i \cdot \left[\frac{\tau^i t(\tau)}{\delta}\right]_1 + s \cdot [A]_1 + r[B]_1 - r \cdot s[\delta]_1 \qquad (12)$$

6) Send proof $\pi = ([A]_1, [C]_1, [B]_2)$

The prover part of the code is end-to-end on device and is verbatim the above using the polynomial API. The high level idea is simple, if you have a polynomial and an operation such as an MSM, feed the pointer to the polynomial data using memory views into the relevant ICICLE API for the curve of interest and perform computations on the data without duplication.

```
//Compute U,V,W from witness, QAP
Polynomial_t U = L_QAP[0].clone();
Polynomial_t V = R_QAP[0].clone();
Polynomial_t W = O_QAP[0].clone();
for (int col = 1; col <= m; ++col) {
    U += witness[col] * L_QAP[col];
    V += witness[col] * R_QAP[col];
    W += witness[col] * O_QAP[col];
}

// ------- PROOF part ----------//
// --- Step 1 --- generate r,s ← F_r
const auto r = S::rand_host();
const auto s = S::rand_host();

// --- Step 2 ---
//compute h(x) = (U(x) * V(x) − W(x))/t(x)
const int vanishing_poly_deg = n;
Polynomial_t h = (U * V -
↪  W).divide_by_vanishing_polynomial(
↪  vanishing_poly_deg);
// config MSM: default: setup data on
↪  device.
auto msm_config =
↪  msm::default_msm_config();
msm_config.are_scalars_on_device =
↪  true;

//  --- Step 3 ---
G1P U_commited;
auto [U_coeff, N, device_id] =
↪  U.get_coefficients_view();
//Compute commitment Σ_i a_i[u_i(τ)]_1
msm::_msm(U_coeff.get(),
↪  pk.g1.powers_of_tau.data(), n,
↪  msm_config, &U_commited);
//compute [A]_1 = Σ_{i=0}^m a_i · [u_i(τ)]_1 + [α]_1 + r · [δ]_1
proof.A = G1P::to_affine(U_commited +
↪  G1P::from_affine(pk.g1.alpha) +
↪  r*G1P::from_affine(pk.g1.delta));
```

```
//  --- Step 4 --- compute [B]_2 and [B]_1
G1P B1;
G2P V_commited_g2;
auto [V_coeff, N, device_id] =
↪  V.get_coefficients_view();
//Compute commitment Σ_i a_i[v_i(τ)]_2
msm::_g2_msm(V_coeff.get(),
↪  pk.g2.powers_of_tau.data(), n,
↪  msm_config, &V_commited_g2);
//Compute  [B]_2 = Σ_{i=0}^m a_i · [v_i(τ)]_2 + [β]_2 + s · [δ]_2
proof.B = G2P::to_affine(V_commited_g2
↪  + pk.g2.beta + s *
↪  G2P::from_affine(pk.g2.delta));

G1P V_commited_g1;
//Compute commitment  Σ_i a_i[v_i(τ)]_1
msm::_msm(V_coeff.get(),
↪  pk.g1.powers_of_tau.data(), n,
↪  msm_config, &V_commited_g1);
//Compute [B]_1 = Σ_{i=0}^m a_i · [v_i(τ)]_1 + [β]_1 + s · [δ]_1
B1 = V_commited_g1 + pk.g1.beta +
↪  G1P::from_affine(pk.g1.delta) * s;

//  --- step 5 --- compute [C]_1
// access quotient from step 1 using
↪  polynomial view
auto [H_coeff, N, device_id] =
↪  h.get_coefficients_view();
G1P HT_commited;
//Compute commitment Σ_i h_i[τ^i t(τ)/δ]_1
msm::_msm(H_coeff.get(), pk.g1.vanishi⌋
↪  ng_poly_points.data(), n - 1,
↪  msm_config, &HT_commited);

G1P private_inputs_commited;
msm_config.are_scalars_on_device =
↪  false;
msm::_msm(witness.data() + l + 1, pk.g⌋
↪  1.private_witness_points.data(), m
↪  - l, msm_config,
↪  &private_inputs_commited));

  //Compute [C]_1 = Σ_{i=l+1}^m a_i · [pk'_i]_1 + Σ_{i=0}^{n−2} h_i ·
↪  [τ^i t(τ)/δ]_1 + s · [π_A]_1 + r[B]_1 − r · s[δ]_1
proof.C = G1P::to_affine(
private_inputs_commited + HT_commited
↪  + G1P::from_affine(proof.A) * s +
↪  B1 * r -r * s *
↪  G1P::from_affine(pk.g1.delta));

// --- step --- 6
return proof;
```

ACKNOWLEDGMENTS

The authors would like to thank Omer Shlomovits and Suyash Bagad for careful review and comments.

REFERENCES

[1] Ingonyama, “ICICLE: GPU Library for ZK Acceleration,” Jan. 2024, https://github.com/ingonyama-zk/icicle.

[2] RISC-V, “Risc-v manual,” https://github.com/riscv/riscv-isa-manual/releases/tag/20240411.

[3] W.-A. community group, “Web assembly manual,” https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[4] J. Groth, “On the size of pairing-based non-interactive arguments,” Cryptology ePrint Archive, Paper 2016/260, 2016, https://eprint.iacr.org/2016/260.

[5] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “Plonk: Permutations over lagrange-bases for oecumenical non-interactive arguments of knowledge,” Cryptology ePrint Archive, Paper 2019/953, 2019, https://eprint.iacr.org/2019/953.

[6] Privacy-Scaling-Explorations, “Halo2 kzg.” [Online]. Available: https://github.com/privacy-scaling-explorations/halo2

[7] “Plonky2,” https://github.com/0xPolygonZero/plonky2.

[8] “Plonky3,” https://github.com/Plonky3/Plonky3.

[9] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, “Scalable, transparent, and post-quantum secure computational integrity,” Cryptology ePrint Archive, Paper 2018/046, 2018, https://eprint.iacr.org/2018/046. [Online]. Available: https://eprint.iacr.org/2018/046

[10] A. Kothapalli, S. Setty, and I. Tzialla, “Nova: Recursive zero-knowledge arguments from folding schemes,” Cryptology ePrint Archive, Paper 2021/370, 2021, https://eprint.iacr.org/2021/370.

[11] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, “Consensys/gnark: v0.9.0,” Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5819104

[12] D. Labs, “zkwasm,” https://github.com/DelphinusLab/zkWasm.

[13] A. Kate, G. M. Zaverucha, and I. Goldberg, “Constant-size commitments to polynomials and their applications,” in *Advances in Cryptology - ASIACRYPT 2010*, M. Abe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 177–194.

[14] C. Papamanthou, E. Shi, and R. Tamassia, “Signatures of correct computation,” Cryptology ePrint Archive, Paper 2011/587, 2011, https://eprint.iacr.org/2011/587.

[15] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, “Bulletproofs: Short proofs for confidential transactions and more,” Cryptology ePrint Archive, Paper 2017/1066, 2017, https://eprint.iacr.org/2017/1066.

[16] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby, “Brakedown: Linear-time and field-agnostic snarks for r1cs,” Cryptology ePrint Archive, Paper 2021/1043, 2021, https://eprint.iacr.org/2021/1043.

[17] R. S. Wahby, I. Tzialla, abhi shelat, J. Thaler, and M. Walfish, “Doubly-efficient zksnarks without trusted setup,” Cryptology ePrint Archive, Paper 2017/1132, 2017, https://eprint.iacr.org/2017/1132.

[18] A. Chiesa and E. Yogev, “Building cryptographic proofs from hash functions,” http://snargsbook.org. [Online]. Available: http://snargsbook.org

[19] U. Haböck, “A summary on the fri low degree test,” Cryptology ePrint Archive, Paper 2022/1216, 2022, https://eprint.iacr.org/2022/1216.

[20] A. Belling, A. Soleimanian, and B. Ursu, “Vortex: A list polynomial commitment and its application to arguments of knowledge,” Cryptology ePrint Archive, Paper 2024/185, 2024, https://eprint.iacr.org/2024/185.

[21] Ingonyama, “Zpu - zero knowledge processing unit,” https://medium.com/@ingonyama/zpu-the-zero-knowledge-processing-unit-f886a48e00e0.

[22] arkworks contributors, “`arkworks` zksnark ecosystem,” 2022. [Online]. Available: https://arkworks.rs

[23] “Pilstark - polygon,” https://github.com/0xPolygonHermez/pil-stark.

[24] A. A. Badawi, A. Alexandru, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, C. Pascoe, Y. Polyakov, I. Quah, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, “OpenFHE: Open-source fully homomorphic encryption library,” Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915.

[25] ZAMA, “Tfhe-rs,” https://github.com/zama-ai/tfhe-rs.

[26] S. Kim, W. Wang, D. Kim, A. Vartak, M. Steiner, and R. Cammarota, “Towards a polynomial instruction based compiler for fully homomorphic encryption accelerators,” Cryptology ePrint Archive, Paper 2024/707, 2024, https://eprint.iacr.org/2024/707.

[27] Google, “Heir: Homomorphic encryption intermediate representation,” https://github.com/google/heir.

[28] V. Shoup, “Ntl – a library for doing numbery theory – version 11.5.1,” https://github.com/libntl/ntl.

[29] C. Aguilar Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, “NFLlib: NTT-based Fast Lattice Library,” in *Cryptographers' Track at the RSA Conference 2016*, ser. Lecture Notes in Computer Science book series (LNCS), K. Sako, Ed., vol. 9610, no. Chapter : Lattice Cryptography, San Francisco, United States, Feb. 2016, pp. 341–356. [Online]. Available: https://hal.science/hal-01242273

[30] S. Halevi and V. Shoup, “Design and implementation of HElib: a homomorphic encryption library,” Cryptology ePrint Archive, Paper 2020/1481, 2020, https://eprint.iacr.org/2020/1481.

[31] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, “Marlin: Preprocessing zksnarks with universal and updatable srs,” Cryptology ePrint Archive, Paper 2019/1047, 2019, https://eprint.iacr.org/2019/1047.

[32] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù, “Gemini: Elastic snarks for diverse environments,” Cryptology ePrint Archive, Paper 2022/420, 2022, https://eprint.iacr.org/2022/420.

[33] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, “Circom: A circuit description language for building zero-knowledge applications,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4733–4751, 2023.

[34] “Zokrates,” https://github.com/Zokrates/ZoKrates.

[35] L. Goldberg, S. Papini, and M. Riabzev, “Cairo – a turing-complete stark-friendly cpu architecture,” Cryptology ePrint Archive, Paper 2021/1063, 2021, https://eprint.iacr.org/2021/1063.

[36] A. Arun, S. Setty, and J. Thaler, “Jolt: Snarks for virtual machines via lookups,” Cryptology ePrint Archive, Paper 2023/1217, 2023, https://eprint.iacr.org/2023/1217.

[37] S. Setty, J. Thaler, and R. Wahby, "Unlocking the lookup singularity with lasso," Cryptology ePrint Archive, Paper 2023/1216, 2023, https://eprint.iacr.org/2023/1216.
[38] N. Amin, J. Burnham, F. Garillot, R. Gennaro, C. Künzang, D. Rogozin, and C. Wong, "Lurk: Lambda, the ultimate recursive knowledge," Cryptology ePrint Archive, Paper 2023/369, 2023, https://eprint.iacr.org/2023/369.
[39] Polygon, "Miden virtual machine," https://github.com/0xPolygonMiden/miden-vm.
[40] B. Threadbare, "Miden assembly," https://hackmd.io/@bobbinth/ry-OIBwPF.
[41] Polygon, "Miden vm," https://github.com/0xPolygonMiden.
[42] D. Marin, M. Abdalla, P. Govereau, J. Groth, S. Judson, K. Sosnin, G. V. Policharla, and Y. Zhang, "Nexus 1.0: Enabling verifiable computation," https://www.nexus.xyz/whitepaper.pdf.
[43] A. Kothapalli and S. Setty, "Hypernova: Recursive arguments for customizable constraint systems," Cryptology ePrint Archive, Paper 2023/573, 2023, https://eprint.iacr.org/2023/573.
[44] Powdr-labs, "Powdr vm," https://github.com/powdr-labs/powdr.
[45] J. Bruestle, P. Gafni, and the RISC Zero Team, "Risc zero zkvm: Scalable, transparent arguments of risc-v integrity," https://dev.risczero.com/proof-system-in-detail.pdf.
[46] RISC0, "Risc zero," https://github.com/risc0/risc0.
[47] Succinct-Labs, "Sp1," https://github.com/succinctlabs/sp1?tab=readme-ov-file.
[48] Valida-XYZ, "Valida zkvm," https://github.com/valida-xyz/valida?tab=readme-ov-file.

APPENDIX A
ZKP FRONTEND

Please refer to tables 3, 4, 5

APPENDIX B
ICICLE COMPILATION

ICICLE is a statically compiled library, clone the ICILE library [1] locally and compile for a specific curve or field

```
git clone https://github.com/ingonyama⌋
↪  -zk/icicle.git --branch main --
↪  single-branch
mkdir -p build
cmake -OPTIONS=ON/OFF -DCURVE=<CURVE>
↪  -S . -B build;
cmake --build build -j
```

Where `<CURVE>` can be one of `bn254/bls12_377/bls12_381/bw6_761/grumpkin`. If compiled in field mode the option `-DFIELD=<FIELD>` currently supports `babybear/stark252` future versions will include `mersenne/goldilocks`. This outputs two statically compiled files

```
libingo_curve_<CURVE>.a
libingo_field_<CURVE>.a
```

into `build/lib`. The compilation options are summarized in table 6. These statically generated files must be linked to the executable in `Cmake` by providing the appropriate path. See below for an example for `bn254`

```
    add_executable(
    <executable_name>
    src/file_1.cu
    src/file_2.cu
)
target_link_libraries(<executable_name>
${CMAKE_SOURCE_DIR}/../icicle/icicle/b⌋
↪   uild/lib/libingo_curve_bn254.a
${CMAKE_SOURCE_DIR}/../icicle/icicle/b⌋
↪   uild/lib/libingo_field_bn254.a
)
target_include_directories(<executable⌋
↪   _name>
↪   PRIVATE
"/../icicle/icicle/include"
"${CMAKE_SOURCE_DIR}/include")
```

APPENDIX C
POLYNOMIALS IN FINITE FIELDS

In this section, we cover some relevant background on polynomials in finite fields. For the purpose of this paper, we limit to univariate polynomials, and exclude polynomials on rings (these will be supported in future versions of ICICLE).

C.1 Basics

Let $\mathbb{F}_r$ be a prime field of characteristic p, a univariate polynomial in $\mathbb{F}_r$ is defined as an expression of the form

$$P(x) = \sum_{i=0}^{n-1} a_i x^i \tag{13}$$

where $a_i \in \mathbb{F}_r \; \forall \; i = 0, 1, \ldots, n-1$ are called coefficients, and $n-1 = deg(P(x)) \geq 0$ is the degree of the polynomial. Thus a polynomial of degree $n-1$ is a tuple $\mathbb{F}_r^n$ in a basis. This representation above is known as the "coefficients form" or the monomial basis. Given a point $c \in \mathbb{F}_r$ then the evaluation of P at b is defined as

$$P(b) = \sum_{i=0}^{n-1} a_i b^i \tag{14}$$

TABLE 3
Front end paradigms ASIC vs CPU approach

Feature	ASIC approach	CPU approach
Program logic	deterministic	non-deterministic
Circuit encoding	hard coded arithmetic circuits	Compile to virtual Machine (VM) Instruction Set Architecture (ISA)
Advantages	Efficiency community templates for circuits Optimizations	Flexiblity Program in high level languages less cryptography knowledge
Disadvantages	Low level languages for circuit Large surface area for bugs	difficult to optimize rely on compilers for ISA
Backend function	prove CSAT	prove ISA execution

TABLE 4
Frameworks that consist of tools to arithmetize circuits in ASIC approach and also provide different backend support for generating ZKPs.

Frontend	Supported Backends
Arkworks [22] (R1CS)	Groth16 [4], Marlin [31], Gemini [32]
Circom [33] (R1CS)	Groth16
Gnark [11] (R1CS/Plonk)	Groth16, Plonk [5] +KZG [13]
Halo2 (Plonkish)	Halo2 IPA , Halo2 KZG [6]
Plonky2/3 [7], [8] (Plonkish)	Plonkish + STARK [9]
ZoKrates [34]	Groth16, Marlin

TABLE 5
Frameworks that follow the CPU approach: VM's with code execution reducible to given ISA and their supported backends

zkVM	ISA	Backend
CAIRO [35]	RISC (Custom)	STARK [9]
Jolt [36]	RISC-V	LASSO [37]
Lurk [38]	Lurk (Custom)	Nova [10]
Miden [39]	Assembly (Custom) [40]	STARK based [41]
Nexus [42]	NVM (Custom)	Nova [10], Hypernova [43]
Powdr [44]	Assembly (Custom)	Halo2KZG [6]
RISC0 [45]	RISC-V	STARK+Groth16 [46]
SP1 [47]	RISC-V	Plonky3 [8]
Valida [48]	RISC (Custom)	Plonky3 [8]

Polynomial arithmetic follows as usual from linear and distributive laws of algebra

$$A(x) + B(x) = \sum_{i=0}^{n_1-1} a_i x^i + \sum_{i=0}^{n_2-1} b_i x^i = \sum_{i=0}^{max(n_1-1,n_2-1)} (a_i + b_i) x^i \tag{15}$$

$$A(x) \cdot B(x) = \left(\sum_{i=0}^{n_1-1} a_i x^i\right) \cdot \left(\sum_{j=0}^{n_2-1} b_j x^j\right) = \sum_{i=0}^{n_1+n_2-2} \sum_{j=0}^{i} a_i b_{i-j} x^i \tag{16}$$

where $\cdot$ is the usual Cartesian product. It follows that

$$deg(A(x)) + deg(B(x)) = deg(A(x)B(x)) \tag{17}$$

$$deg(A(x) + B(x)) \leq max(deg(A(x), B(x))) \tag{18}$$

We summarize some basic properties of polynomials in $\mathbb{F}_r$ relevant for us below.

- A polynomial has a multiplicative inverse only if the degree of the polynomial is zero (constant polynomial). [13]
- if $A(x) \cdot B(x) = 0$ either $A(x) = 0$ or $B(x) = 0$
- if $A(x) \neq 0$ and $A(x) \cdot B(x) = A(x) \cdot C(x)$ then $B(x) = C(x)$.

13. If $A(x) \cdot B(x) = 1$ then $deg(A(x)) + deg(B(x) = 0$, it follows that $A(x) = c, B(x) = c^{-1}$ for $c \in \mathbb{F}_r$

TABLE 6
ICICLE compilation options

Option	mode	Default
`EXT_FIELD`	Extension field (field mode)	`OFF`
`DMSM`	MSM (curve mode)	`ON`
`DBUILD_HASH`	Hashes (any mode)	`OFF`
`DG2`	$\mathbb{G}_2$ EC arithmetic (curve mode)	`OFF`
`DECNTT`	ECNTT (curve mode)	`OFF`
`BUILD_TESTS`	test binary (any mode)	`OFF`
`DBUILD_BENCHMARK`	bench suite (any mode)	`OFF`
`DEVMODE`	debug mode	`OFF`

- For $B(x) \neq 0$, there exists unique quotient and remainder polynomials $Q(x), R(x)$ with $deg(R(x)) < deg(B(x))$ such that

$$\begin{aligned} A(x) &= B(x) \cdot Q(x) + R(x) \\ R(x) &\equiv A(x) \mod B(x) \end{aligned} \tag{19}$$

and can be computed using the Euclidean division algorithm.
- For $\beta \in \mathbb{F}_r$, $A(x) = (x - \beta) \cdot Q(x) + A(\beta)$ and in particular if $A(x)$ has a factor $(x - \beta)$, then β is a root of $A(x) \in \mathbb{F}_r$.
- Lagrange interpolation: Given distinct $a_i \in \mathbb{F}_r^n$ (basis), and $b_i \xleftarrow{\$} \mathbb{F}_r^n$, there exists a unique polynomial $P(x)$ with $deg(P(x)) \leq n-1$ such that

$$P(x) = \sum_{i=0}^{n-1} b_i \cdot L_i(x) \ ; \ L_i(x) = \prod_{\substack{k=0 \\ k \neq i}} \left(\frac{x - a_k}{a_i - a_k} \right) \tag{20}$$

where $L_i(x)$ are the Lagrange bases that satisfy $L_i(a_j) = \delta_{ij}$. This representation of a polynomial is known as the "evaluations form" or the Lagrange basis, since we represent $b_i \equiv P(a_i)$. Since $L_i(a_j) = \delta_{ij}$ the multiplication operation of two polynomials (of n evaluations each) in evaluation form is Hadamard multiplication (elementwise)

$$\begin{aligned} P(x) \cdot Q(x) =& \sum_{i=0}^{n-1} p_i \cdot L_i(x) \sum_{j=0}^{n-1} q_j \cdot L_j(x) \\ =& [p_0 q_0, p_1 q_1, \ldots, p_{n-1} q_{n-1}] \\ \equiv& \vec{p}_n \circ \vec{q}_n \end{aligned} \tag{21}$$

An auxiliary identity that follows from the above is that if in **evaluations** form the identity $P(x) \cdot Q(x) = R(x)$ holds then

$$\sum_{i=0}^{n-1} p_i \cdot L_i(x) \sum_{j=0}^{n-1} q_j \cdot L_j(x) = \sum_{k=0}^{n-1} r_k \cdot L_k(x) \tag{22}$$

it implies $q_j = r_j / h_j \ \forall j \in \{0, 1, \ldots, n-1\}$, when there are no zero divisors.

C.2 Polynomials in $\mathbb{F}^{\times}$

In any finite field $\mathbb{F}_r$ the non zero elements $\mathbb{F}_r^{\times} = \mathbb{F}_r - \{0\}$ form a multiplicative group. Since every multiplicative group/subgroup in a finite field is cyclic, it follows that for every $x \in \mathbb{F}_p^{\times}$, we have $x^{p-1} = 1$, or x is a pth root of unity. In practice, if we have n field elements b_i, it is sufficient to represent this as a polynomial by choosing a multiplicative subgroup domain $H_n \subset \mathbb{F}_r^{\times}$. This domain can be constructed by computing $\omega = x^{\frac{p-1}{n}}$, such that $\omega^n = 1$ i,e ω is a nth root of unity in $\mathbb{F}_r$. If in addition $\omega^{n/2} \neq 1$ it is a primitive root.[14] Note that finding a root of unity is equivalent to finding a solution of the following equation in $\mathbb{F}_r$

$$t(x) = x^n - 1 = 0 \tag{23}$$

Thus the n solutions (roots of unity) to this equation form a domain

$$H_n \equiv \{1, \omega, \omega^2, \ldots \omega^{n-1}\} \tag{24}$$

The equation (23) is called a vanishing polynomial in the literature, it is unique and defines the domain on which other polynomials upto degree $n-1$ can be interpolated. **We use the notation H_d to mean that the domain consists of d points and can atmost represent evaluations of a polynomial of degree**

14. The number of primitive roots in $\mathbb{F}_r$ is $\phi(p-1)$ where Φ is the Euler Totient Function.

$d-1$. We can represent a polynomial in either the coefficient or the evaluation form in the domain H_n

$$P(x) = \sum_{i=0}^{n-1} c_i \cdot x^i \quad \text{(Coefficient form)} \tag{25}$$

$$P(x) = \sum_{i=0}^{n-1} e_i \cdot L_i(x) \quad \text{(Evaluation form)} \tag{26}$$

with $e_i = P(\omega^i)$ and can be defined explicitly using Lagrange interpolation (20) defined on H_n. The relationship between coefficients c_i and evaluations e_i is more transparently expressed (and practical) using the Discrete Fourier Transform defined on $H_n \subset \mathbb{F}_r^x$ which is referred to commonly as the NTT (Number Theoretic Transform). The transformation matrix is defined as

$$F_n = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)^2} \end{pmatrix}_{n\times n} \tag{27}$$

For notational convenience we define the vector of coefficients as $\vec{c}_n$ and vector of evaluations as $\vec{e}_n$, for polynomial of degree $n-1$. In general we define the NTT_n as the operation that takes polynomial in coefficients representation (n points) to the evaluation representation and $INTT_n$ as the operation that takes a polynomial from the evaluation to the coefficients representation

$$NTT_n \ : \ \vec{e}_n = F_n \cdot \vec{c}_n \quad \text{(coeffs to evals)} \tag{28}$$

$$INTT_n \ : \ \vec{c}_n = F_n^{-1} \cdot \vec{e}_n \quad \text{(evals to coeffs)} \tag{29}$$

The computation of the NTT/INTT matrix follows the usual DFT (Discrete Fourier Transform) complexity of $\mathcal{O}(n \log n)$ and is efficiently evaluated using well known DFT algorithms. In this language, interpolation of a polynomial of degree n to say degree $m-1 > n$, means first to define $H_m \equiv \{1, \omega_m, \dots, \omega_m^{m-1}\}$ where $\omega^m = 1$ in $F_p^\times$ and do an NTT_m

$$NTT_m \ : \ \vec{e}_m = F_m \cdot [\vec{c}_n || \vec{0}_{m-n}] \tag{30}$$

Closely following the interpolation rule is the convolution rule. Given two polynomials $P_1(x) = \sum_{i=0}^{n_1-1} p_i x^i, P_2(x) = \sum_{i=0}^{n_2-1} q_i x^i$ defined in $H_{max(n_1-1,n_2-1)} \subset \mathbb{F}_r^\times$. From (17) the result is of degree $d = n-1 = n_1+n_2-2$, hence we define the product domain $H_n \equiv \{1, \omega_n, \dots, \omega_n^{n-1}\}$ with $\omega_n^n = 1$ and

$$\begin{aligned} P_1(x)\cdot P_2(x) &= F_n^{-1} \cdot \Big((F_n \cdot [\vec{p}_{n_1} || \vec{0}_{n_2-1}])^t \\ &\qquad \circ (F_n \cdot [\vec{p}_{n_2} || \vec{0}_{n_1-1}]) \Big) \\ &= INTT_n \left(NTT_n(P_1(x)) \circ NTT_n(P_2(x)) \right) \end{aligned} \tag{31}$$

Thus at the end of the multiplication, the $INTT_n$ returns the result in coefficients form. However, if one needs to perform further polynomial manipulations, such as addition, multiplication, or division it is advisable to retain the polynomial in evaluation form as long as the degree requirements are met for the final result.

C.3 Polynomial Identity checking with quotient argument

Polynomial identity checking is a central instrument in PIOP/PCP of ZKPs.. To check a statement $P(x) \overset{?}{\equiv} Q(x)$ in some domain H_n, it is equivalent to check the statement $R(x) \overset{?}{\equiv} 0$ in H_n, where $R(x) = P(x) - Q(x)$, with $deg(R(x)) = d$. Thus, if $R(x) \overset{?}{\equiv} 0$ in H_n, it follows from fundamental theorem of algebra that it must be proportional to the vanishing polynomial (23) i.e

$$R(x) \overset{?}{\equiv} h(x) \cdot t(x) \tag{32}$$

where $deg(h(x)) = m-1 = d-n$. Therefore, we can state that $R(x)$ vanishes in H_n iff there exists a polynomial $h(x)$ of degree $d-n$ such that (32) holds. In practice, since both $t(X)$ and $R(x)$ vanish in H_n in (32), we cannot compute the quotient using high-school multiplication. An efficient method to compute the quotient is to recognize that the in order to adequately represent the quotient of degree $m-1$, we need a domian of atleast m points. This is achieved by defining a coset domain $H_{m,\eta} = \eta \cdot \{1, \omega_m, \dots, \omega_m^{m-1}\}$[15] for some $\eta \in \mathbb{F}_r$, and $\eta \notin H_n$. Note that the vanishing polynomial (23) evaluates to a constant $t(x)\big|_{H_{m,\eta}} = (\eta^m - 1)[\vec{1}_m]$ in $H_{m,\eta}$. Thus $h(x)$ can be efficiently computed in coefficients form as

$$h(x) = \frac{ICosetNTT_{H_{m,\eta}}\left([\vec{1}_m] \circ CosetNTT_{H_{m,\eta}}(R(x))\right)}{(\eta^m - 1)} \tag{33}$$

15. This is known as coset NTT, i.e the NTT matrix (27) is now defined with the modified roots of unity in the coset domain.

As a concrete example, in the Groth16 QAP quotient argument [4] (§4) we have $deg(A) = deg(B) = deg(C) = N - 1$ all defined in H_N. The quotient argument is

$$h(x) = \frac{A(x) \cdot B(x) - C(x)}{x^N - 1} \tag{34}$$

where the numerator $R(x) = A(X) \cdot B(x) - C(x)$ is of $deg(R(x)) = 2N - 2$, and the degree of the quotient is $deg(h(x)) = m - 1 = 2N - 2 - N = N - 2$. Thus we need to define a coset H_m of $m = N - 1$ points to adequately represent the $N - 1$ evaluations of $h(x)$

$$\begin{aligned} h(x) = ICosetNTT_{H_{m,\eta}} \Big(& CosetNTT_{H_{m,\eta}}(A(x)) \\ & \circ CosetNTT_{H_{m,\eta}}(B(x)) \\ & - CosetNTT_{H_{m,\eta}}(C(x)) \Big) \times \frac{1}{\eta^m - 1} \end{aligned} \tag{35}$$

The quotient argument as defined above is generalizable for arbitrary polynomial identities on $\mathbb{F}_r^{\times}$.